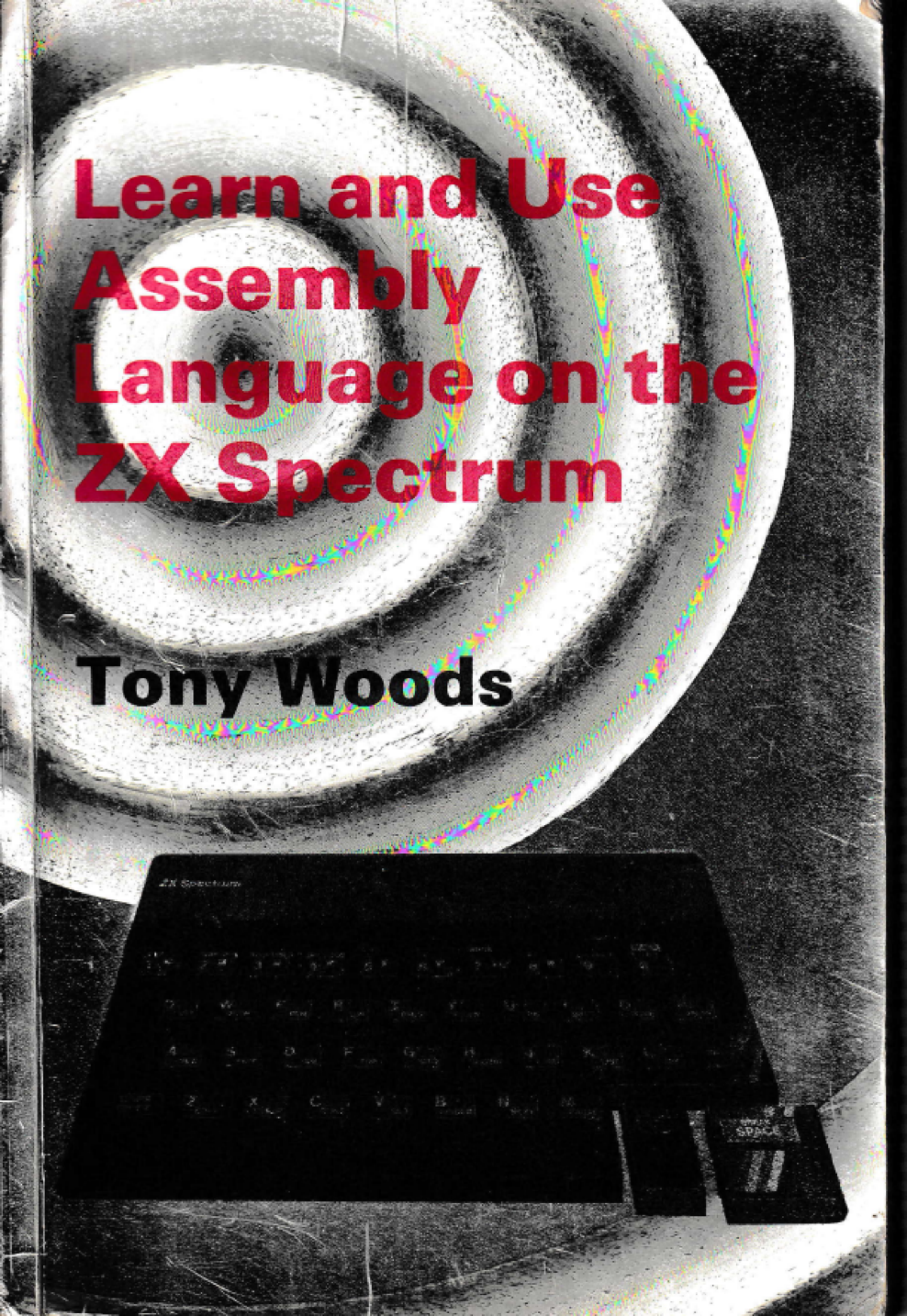

ZX Spectrum
SPACE

Learn and Use Assembly Language on the ZX Spectrum

Tony Woods

LEARN AND USE ASSEMBLY LANGUAGE ON THE ZX SPECTRUM

Learn and use assembly language on the ZX Spectrum

Tony Woods

McGRAW-HILL Book Company (UK) Limited

London · New York · St Louis · San Francisco · Auckland · Bogotá
Guatemala · Hamburg · Johannesburg · Lisbon · Madrid · Mexico
Montreal · New Delhi · Panama · Paris · San Juan · São Paulo
Singapore · Sydney · Tokyo · Toronto

Published by
McGRAW-HILL Book Company (UK) Limited
MAIDENHEAD · BERKSHIRE · ENGLAND

British Library Cataloguing in Publication Data

Woods, Tony
Learn and use assembly language on the ZX Spectrum.
1. Sinclair ZX Spectrum (Computer)—Programming
2. Assembler language (Computer program language)
I. Title
001.64'25 QA76.8.S625

ISBN 0-07-084705-3

Library of Congress Cataloging in Publication Data

Woods, Tony
Learn and use assembly language on the ZX Spectrum.
1. Sinclair ZX Spectrum (Computer)—Programming.
2. Assembler language (Computer program language)
I. Title
QA76.8.S623W66 1983 001.64'2 83-13550
ISBN 0-07-084705-3

Copyright © 1983 McGraw-Hill Book Company (UK) Limited. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior permission of McGraw-Hill Book Company (UK) Limited, or of the original copyright holder.

1 2 3 4 5 6 CUP 8543

Typeset by Oxprint Ltd, Oxford
Printed and Bound in Great Britain
by Cambridge University Press

CONTENTS

Preface xi

Chapter 1 Inside the Spectrum 1

1.1 The spectrum computer 1
1.2 Computer languages 2
1.3 The Z80 microprocessor 4
1.4 The registers 6

Chapter 2 Bits and bytes 7

2.1 Binary numbers 7
2.2 Changing number bases 9
2.3 Bits and bytes 10
2.4 Memory 11

Chapter 3 Assembly language programming 13

3.1 Assembly language 13
3.2 An example program 15
3.3 Instructions 15
3.4 Assembling 17
3.5 Program storage 19
3.6 Subroutines 20

Chapter 4 Some simple instructions 24

4.1 Data in the computer 24
4.2 Loading registers 24
4.3 Increment and decrement 26
4.4 Memory transfers 26
4.5 Addition and subtraction 28
4.6 Writing a program 29
4.7 Labels 31
4.8 Program 32

Chapter 5 Jumping around 33

5.1 Why jump? 33
5.2 Unconditional jumps 33

5.3 The flag register 34
5.4 Conditional jumps 35
5.5 Comparisons 37
5.6 Pseudo-operations 39
5.7 Screen output 40
5.8 Program 44

Chapter 6 Using the keyboard 46

6.1 Keyboard input 46
6.2 Character codes 47
6.3 Number input 47
6.4 Negative numbers 50
6.5 Carry and overflow 51
6.6 The EQU instruction 55
6.7 Program 56

Chapter 7 Sixteen-bit numbers 57

7.1 Register pairs 57
7.2 Sixteen-bit data in memory 57
7.3 More from the keyboard 58
7.4 Computer music 61
7.5 Addressing modes 61
7.6 Program 65

Chapter 8 Doing it again 66

8.1 Loops 66
8.2 Counting loops 67
8.3 What is a stack? 70
8.4 Uses of stacks 72
8.5 Displayed messages 74
8.6 Nested loops 76
8.7 Program 77

Chapter 9 The display 78

9.1 The display file 78
9.2 Finding the characters 79
9.3 Block moves 81
9.4 Some display routines 84

9.5 The screen border 86
9.6 Program 87

Chapter 10 Multiplication and division 88

10.1 Shift instructions 88
10.2 Multiplication 89
10.3 Rotations 90
10.4 Program 92

Chapter 11 Bits of logic 93

11.1 Bit operations 93
11.2 Logic instructions 94
11.3 Packed data 96
11.4 Packing and unpacking data 97
11.5 The attribute file 99
11.6 Program 100

Chapter 12 Blocks and tables 101

12.1 Block searches 101
12.2 Index registers 102
12.3 Look-up tables 103
12.4 Jump tables 110
12.5 Random numbers 113
12.6 Program 115

Chapter 13 More arithmetic 116

13.1 Sixteen-bit numbers 116
13.2 Multiple-byte numbers 118
13.3 Binary coded decimal 119
13.4 BCD arithmetic 120
13.5 Other instructions 123
13.6 Program 124

Chapter 14 Ordering the data 125

14.1 Sorting data 125
14.2 Bubble sort 125
14.3 Shell sort 125

Appendix A Summary of assembly language instructions 132

A.1 Flag register 132
A.2 Eight-bit load instructions 134
A.3 Sixteen-bit load instructions 135
A.4 PUSH and POP instructions 136
A.5 Exchange instructions 136
A.6 Block instructions 136
A.7 General arithmetic 136
A.8 Eight-bit arithmetic and logic 137
A.9 Sixteen-bit arithmetic 137
A.10 Rotate and shift instructions 138
A.11 Bit instructions 139
A.12 Jump, call, and return instructions 142
A.13 Restart instructions 142
A.14 Input and output instructions 143
A.15 Miscellaneous instructions 143

Appendix B The ZX Spectrum machine code assembler 144

B.1 Using an assembler 144
B.2 ZX Spectrum assembler 144
B.3 Directives 146

Appendix C Hexadecimal - decimal conversion tables 147

C.1 Conversion of hexadecimal numbers up to FF, or 255 in decimal 147
C.2 Conversion of hexadecimal numbers up to FFFF, or 65 535 in decimal (in conjunction with Table C.1) 148

Appendix D Hand assembly 149

D.1 General method 149
D.2 Addresses and data 149
D.3 Jump instructions 150
D.4 Bit instructions 150
D.5 Index registers 151

Appendix E Character codes 152

Appendix F Print control characters 154

Appendix G ROM subroutines 155

G.1 The ROM program 155
G.2 Print a character 155
G.3 Clear screen 156
G.4 Screen scrolling 156
G.5 Border colour 156
G.6 Screen colours 156
G.7 Keyboard input 157
G.8 Sound 157
G.9 The printer 157
G.10 Graphics 158

Index 159

PREFACE

As you are reading this, you may possibly be feeling a little dissatisfied with what your Spectrum can do using BASIC. Assembly language programs allow you to control the microprocessor, the brain of your Spectrum, directly. You have total control of all the facilities built into your computer.

There are two main reasons for writing programs in assembly language instead of BASIC; first, you will gain tremendously in processing speed and, second, your program will occupy much less memory.

The increase in speed is difficult to appreciate until you have seen it. It will enable you to have smooth-moving graphics at the speed you want them to move. Normally, moving graphics will need slowing down to give a suitable display. Under normal circumstances an assembly language program will be at least twenty times faster than the same program in BASIC and it can be up to 200 times faster.

The reduction in memory used by an assembly language program means that you can hold more data in memory for your program to use. A further advantage of assembly language is that it also gives you complete control over the way data are stored. Using advanced storage techniques you can, in certain circumstances, reduce the space required for data storage to a fraction of the space required by BASIC.

This book shows you how to write programs in assembly language. Before an assembly language program can be run it must be translated into machine code, which is the only language understood directly by the computer. Since a machine code program consists entirely of a series of numbers, it is impracticable to write a program directly in machine code. Assembly language is the nearest, easily understood, language to machine code.

The translation from assembly language to machine code is best carried out, on the computer, using a program called an assembler. It can be carried out manually, but for all except the smallest of programs it is very tedious and liable to mistakes. There are several assembler programs available for the Spectrum; all the programs in this book were produced using the ZX Spectrum Machine Code Assembler. This program has all the facilities needed for producing and translating assembly language programs. It is, in addition, easy to use. It is the ideal program for the newcomer to assembly language as well as providing the facilities needed by the experienced assembly language programmer.

Assembly language is simply another programming language and you should find it is no harder to learn than BASIC. As with all programming languages the only way to learn it is to write plenty of programs. At the end of each chapter I have given a programming problem for the reader to attempt. Each program is designed to be within the capabilities of the reader and yet to demand some thought.

Finally, my thanks go to my wife, Marilyn, for typing the text and supporting me during the writing. Thanks also to my son, Richard, for his help in entering and debugging the example programs.

TONY WOODS

1 INSIDE THE SPECTRUM

1.1 The Spectrum computer

The Spectrum is a powerful personal and home computer. It can be used for a wide variety of applications; it can entertain, it can teach, it can even be used to run a business.

Most uses of the computer have three separate stages:

1. Putting information into the computer
2. Processing that information
3. Displaying the results of the processing

Figure 1.1 shows a typical Spectrum system. Most information is put into the computer using the keyboard, but it can also come

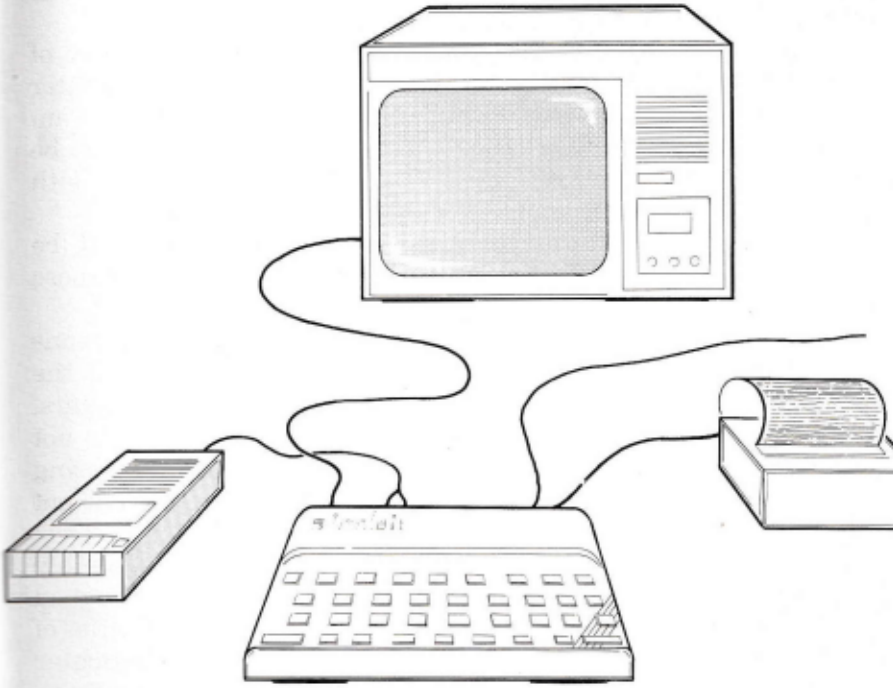


Figure 1.1

from other sources such as games paddles, temperature sensors, or even information which has previously been saved on cassette tape or microdrive. When the information is entered into the computer it is stored or saved in the computer's internal memory.

The next stage, after input, is the processing of the information to produce the required results. This is carried out by a part of the computer known as the central processing unit. The central processing unit of the Spectrum is on a single silicon chip and is called a microprocessor. The microprocessor used in the Spectrum is a Z80A. It is the same microprocessor as used in many expensive business microcomputers.

Finally, the results of the processing are output from the computer. Usually the results of the processing are output to the screen of a television but they could also be in the form of sound from a loudspeaker, printed output from a printer, information saved on cassette tape or microdrive, or even signals to control external devices, such as the valves of a central heating system.

1.2 Computer languages

Before a computer can be used for any application, it must be given instructions to tell it exactly what to do. These instructions are called a *program*.

There are many different programming languages, most of which are designed to be suitable for a particular type of computer application. As an example, Fig. 1.2 shows a small program written in the COBOL language. This language is meant to be used for business programs and its main feature is the ease with which it can be read by people who are not programmers.

Another programming language with which you will be familiar is BASIC. BASIC was designed as a general purpose language which would be easy to learn and use.

Both COBOL and BASIC are *high-level languages*. This means they were designed to suit a particular type of use and the programs produced would run on a wide range of computers. Because they are not tied to a particular computer, they do not make full use of all the facilities built into the central processing unit of the computer; in the Spectrum this means that they do not fully use the capabilities of the Z80 microprocessor.

This book tells you how to program in the Z80 assembly language. This programming language is specifically designed for the Z80 microprocessor and makes use of all the capabilities of the Z80 microprocessor. Because it is designed for a particular type of central processing unit (or microprocessor) it is known as a *low-level programming language*.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
     SELECT TRANSACTION-FILE ASSIGN TO DISK.
     SELECT PRINTFILE ASSIGN TO $LPT.
DATA-DIVISION.
FD TRANSACTION FILE.
DATA RECORD IS TRANS-REC.
01 TRANS-REC.
     02 DEPT             PICTURE 99999.
     02 AMOUNT           PICTURE 99999V99.
FD PRINTFILE.
DATA RECORD IS PRINTLINE.
01 PRINTLINE.
     02 TEXT         PICTURE X(15).
     02 PDEPT        PICTURE 99999.
     02 PSUM         PIICTURE 99999.99.
WORKING-STORAGE SECTION.
77   CURRENT         PICTURE 99999.
77   SUM             PICTURE 99999V99.
PROCEDURE DIVISION.
     MOVE 'TOTAL FOR DEPT.' TO TEXT.
     OPEN INPUT TRANSACTION-FILE.
     OPEN OUTPUT PRINTFILE.
     READ TRANSACTION-FILE.
INIT.
     MOVE DEPT TO CURRENT.
     MOVE AMOUNT TO SUM.
LOOP.
     READ TRANSACTION-FILE.
     IF DEPT NOT EQUAL TO CURRENT GOTO PRINT.
     ADD AMOUNT TO SUM.
     GOTO LOOP.
PRINT.
     MOVE SUM TO PSUM.
     MOVE CURRENT TO PDEPT.
     WRITE PRINTLINE.
     IF DEPT NOT EQUAL TO '99999' GOTO INIT.
     CLOSE TRANSACTION-FILE.
     CLOSE PRINTFILE.
```

Figure 1.2

1.3 The Z80 microprocessor

All microcomputers use a central processing unit wh contained on a single silicon chip. This type of central proc unit is known as a *microprocessor*. There are many ty microprocessors, each having their own assembly langua this book we are only concerned with the Z80 assembly lan which is used by the Spectrum. Because we will be usir capabilities of the microprocessor, directly, we must look internal structure of this central processing unit.

Figure 1.3 shows the general layout of the micropro Basically it consists of a number of different sections whi connected together by an eight-bit data bus. This mean eight-digit binary numbers can be moved and processed l microprocessor.

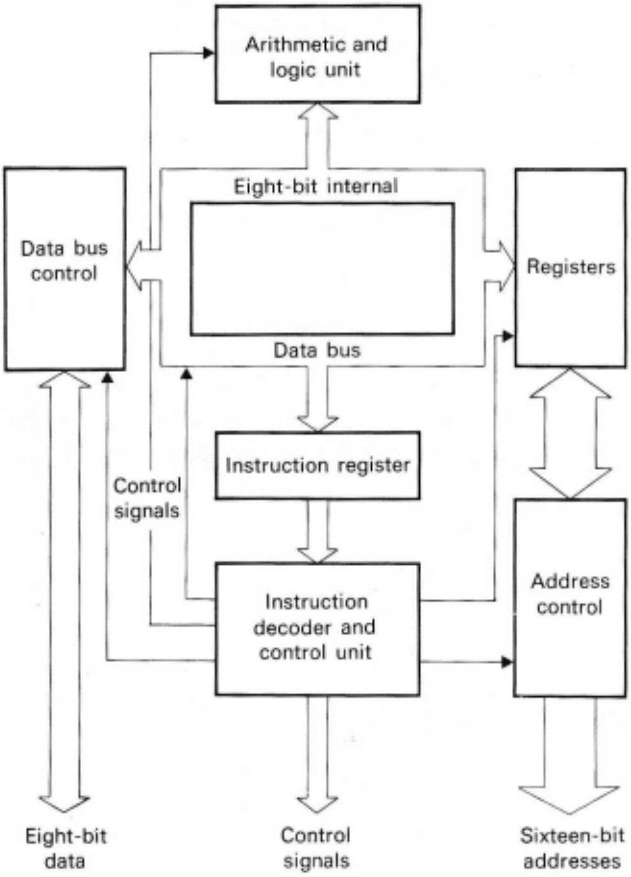

Arithmetic and logic unit
Eight-bit internal
Data bus
Data bus control
Registers
Instruction register
Control signals
Instruction decoder and control unit
Address control
Eight-bit data
Control signals
Sixteen-bit addresses

Figure 1.3

The most important components of the Z80 to the programmer are the registers. Figure 1.4 shows the layout of the registers in the Z80. A register is a memory area which is capable of holding a single item of information. The registers are used to hold data on a temporary basis while they are being processed or waiting to be processed.

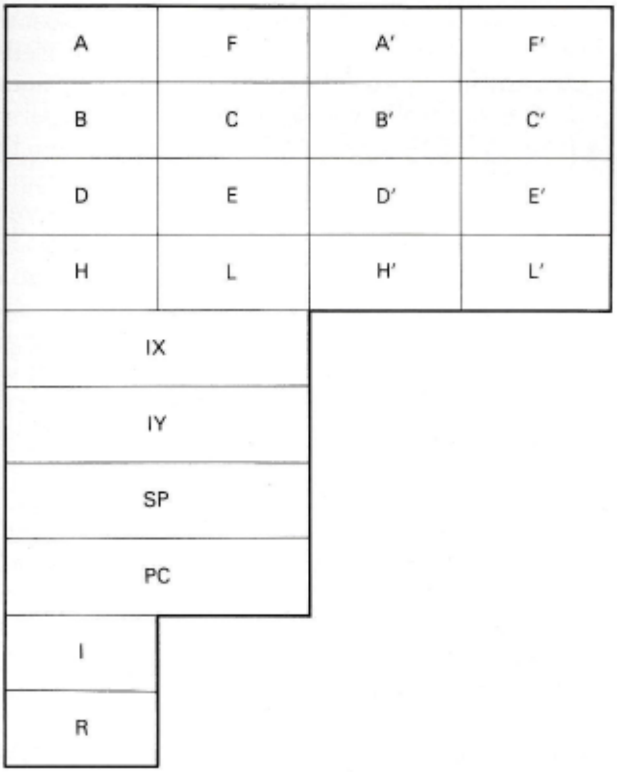


Figure 1.4

All data or information is stored and used inside the computer as binary numbers. For those who are not familiar with binary numbers, they will be explained in the next chapter. Because registers are the storage areas of the computer, they hold data as binary numbers. The Z80 microprocessor has some registers which hold eight-figure, or eight-digit, binary numbers and some which hold 16-digit binary numbers. When talking about binary numbers the term *bit* is used as an abbreviation for 'binary digits'. We call registers which hold eight-digit binary numbers eight-bit registers and registers which hold 16-digit binary numbers 16-bit registers.

The Z80 is an eight-bit microprocessor which means that most of the data used consist of eight-bit numbers. Because the Z80 is an advanced eight-bit microprocessor it also includes some facilities for processing 16-bit numbers.

1.4 The registers

The A register, also called the accumulator, is the most important of the registers. It is an eight-bit register and is used for most arithmetic calculations and other processing operations such as comparisons. For example, when adding two numbers together one of the numbers is first put into the A register, then the second number is added to it, and the result is left in the A register.

The F register, called the flag or status register, is a very special register. It is used to indicate that various conditions have been produced by the processing that has taken place. As an example, after a calculation has been performed, such as the simple addition in the previous paragraph, parts of the F register will indicate whether the result was positive or negative, or whether it was zero or non-zero. These can then be tested by instructions in the program, similar to the IF statement in BASIC. The F register is not used directly by the programmer.

The registers B, C, D, E, H, and L are all general purpose eight-bit registers; they can also be used in pairs, called BC, DE, and HL, as 16-bit registers. Although they are general purpose and can be used virtually interchangeably by conventional use they tend to be used for specific tasks. These uses will be indicated at the appropriate time; however, as an example the HL register pair is often used as a pointer to indicate a specific place in the computer's memory.

The PC register is a 16-bit register whose sole purpose is to indicate to the computer where in memory the next program instruction can be found.

The SP register is a 16-bit register which is used to implement a stack. This is a useful programming device which will be explained later.

The two registers IX and IY are 16-bit registers and are called index registers. They are used when the programmer wishes to use tables or lists of data. The IY register should not be used at all by the programmer on the Spectrum, since it is used by the Spectrum system.

Finally, there are two eight-bit registers, I and R, which have very special uses in the running of the computer; they are only rarely used by programmers.

2 BITS AND BYTES

2.1 Binary numbers

Inside the computer all data consists of sets of electrical pulses. For ease of writing we say that the presence of an electrical pulse can be represented by the digit 1 and the absence of a pulse can be represented by the digit 0. This means that all the data in the computer can be represented by numbers consisting of just zeros and ones. Numbers using just zeros and ones are called *binary numbers.*

Before taking a look at binary numbers it is useful to reconsider the ideas we use with decimal numbers in our everyday life. For example, the number 764 uses the three digits 7, 6, and 4, but because they are in the order shown we know that the seven stands for seven hundred, the six stands for six tens, and the four stands for four ones. We can write this as:

$$764 = 7 \times 100 + 6 \times 10 + 4 \times 1$$

or as:

$$764 = 7 \times (10 \times 10) + 6 \times (10) \times 4 \times (1)$$

This means that each place to the left is ten times more than the previous place. Decimal numbers use a multiplier of 10 for each place, another name for decimal numbers being base 10 numbers.

Although 10 is used as the base for decimal numbers any number can be used as a base. The base numbers often used with computers are base 2, known as binary numbers, and base 16, known as hexadecimal numbers. Binary numbers are the most important for computers because all data in the computer are in the form of binary numbers. Hexadecimal numbers are used because they give an easy shorthand way of writing binary numbers.

Binary and hexadecimal numbers can be broken down into separate digits in the same way as a decimal number. We can write the binary number 1011B as:

$$1011B = 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$$

or as:

$$1011\text{B} = 1 \times (2 \times 2 \times 2) + 0 \times (2 \times 2) + 1 \times (2) + 1 \times (1)$$

and the hexadecimal number 945H as

$$945\text{H} = 9 \times 256 + 4 \times 16 + 5 \times 1$$

or as:

$$945\text{H} = 9 \times (16 \times 16) + 4 \times (16) + 5 \times (1)$$

The letter B at the end of the binary number is used to show that the number is binary and not any other number base. Similarly, the H at the end of the hexadecimal number is used to show that the number is hexadecimal (the ZX Spectrum Assembler uses the common convention of putting a dollar sign in front of the number to show that it is hexadecimal, e.g., $1234 is the same as 1234H).

Any decimal number can be written using the digits from 0 to 9, i.e., by using digits from zero up to one less than the value of the base. In any number base we need symbols for the numbers from zero up to one less than the value of the base.

For binary numbers, using only the digits 0 and 1 we are able to write any binary number. Hexadecimal numbers need symbols for digits 0 up to a value of 15. For the values from 0 up to 9 we use the same symbols used for decimal numbers, i.e., the digits 0 to 9, but for the values 10 to 15 we cannot use the same as decimal numbers because it would be too confusing; we use the letters A to F to stand for the values 10 to 15. Figure 2.1 shows the equivalent binary, hexadecimal, and decimal numbers up to 15.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Figure 2.1

2.2 Changing number bases

Binary and hexadecimal can be converted into decimal by writing them out in full, as shown at the beginning of this section, and then working out their values. Figure 2.2 gives examples of the conversion for a binary and a hexadecimal number.

$$\begin{aligned}
01011011\text{B} &= 1 \times 1 + 1 \times 2 + 0 \times 4 + 1 \times 8 + 1 \times 16 + 0 \times 32 + 1 \\
&\quad \times 64 + 0 \times 128 \\
&= 1 + 2 + 0 + 8 + 16 + 0 + 64 + 0 \\
&= 91
\end{aligned}$$

$$\begin{aligned}
5\text{AC}7\text{H} &= 7 \times 1 + \text{C} \times 16 + \text{A} \times 256 + 5 \times 4096 \\
&= 7 \times 1 + 12 \times 16 + 10 \times 256 + 5 \times 4096 \\
&= 7 + 192 + 2560 + 20480 \\
&= 23239
\end{aligned}$$

Figure 2.2

Converting numbers from decimal to binary is a slightly more difficult task; it involves continually dividing the decimal number by 2 and noting the remainder at each stage. Figure 2.3 shows this process for the decimal number 245. Note that the binary number is found by reading the remainders from the bottom upwards.

$$\begin{array}{l}
2)245(1 \\
2)122(0 \\
2)61(1 \\
2)30(0 \\
2)15(1 \\
2)7(1 \\
2)3(1 \\
2)1(1 \\
\quad 0
\end{array}$$

Figure 2.3

Conversion from decimal to hexadecimal is similar to the decimal-to-binary conversion except that it involves continuing division by 16.

Numbers can be converted from binary to hexadecimal by dividing the binary number into groups of four bits, starting from the right-hand end of the number, extra zeros being added to the left-hand end if necessary to make a final group of four bits. Each four-bit group is then converted into its equivalent hexadecimal digit as shown in Fig. 2.1.

Conversion from hexadecimal to binary is carried out by replacing each hexadecimal digit by its equivalent four-bit group from Fig. 2.1. Figure 2.4 shows examples of these conversions.

0011	1011	0111	1010
3	B	7	A

0011101101111010B = 3B7AH

5	A	C	7
0101	1010	1100	0111

5AC7H = 0101101011000111B

Figure 2.4

2.3 Bits and bytes

The basic data unit in the Spectrum is called a *byte*; a byte is an eight-bit binary number. The value of a byte can be used to represent several different things, such as numbers, characters, or even program instructions. It is important to realize that the same binary number can represent any one of these things depending upon its position in memory. The representation of characters will be dealt with in Chapter 5.

Number representation means that the contents of the byte are considered as a binary number. Therefore a byte which contains the binary digits 01101101 represents the number 1101101B, which is 109 in decimal. The range of numbers which can be held in a single byte is 00000000B to 11111111B, which is 0 to 255 in decimal. This method can only be used to represent positive numbers; it is known as unsigned number representation to distinguish it from the next method, which is used to represent both positive and negative numbers.

Often we will need to use numbers which have negative or positive values. To do this a different method of representation is used; it is called 'two's complement form', or signed numbers. The basis of the method is that in the computer the number of digits in the binary form of any number is always the same. The number of digits is determined by you, the programmer, to give a sufficient range for the problem. The number of digits must be a multiple of eight and often the number of binary digits is eight, which is, of course, a byte.

By this method every bit position in the number is given its normal value except for the extreme left-hand bit which has the negative of its normal value. Figure 2.5 shows the value of each bit in an eight-bit number for both unsigned and signed numbers.

In the signed number the left-hand digit positions represent the

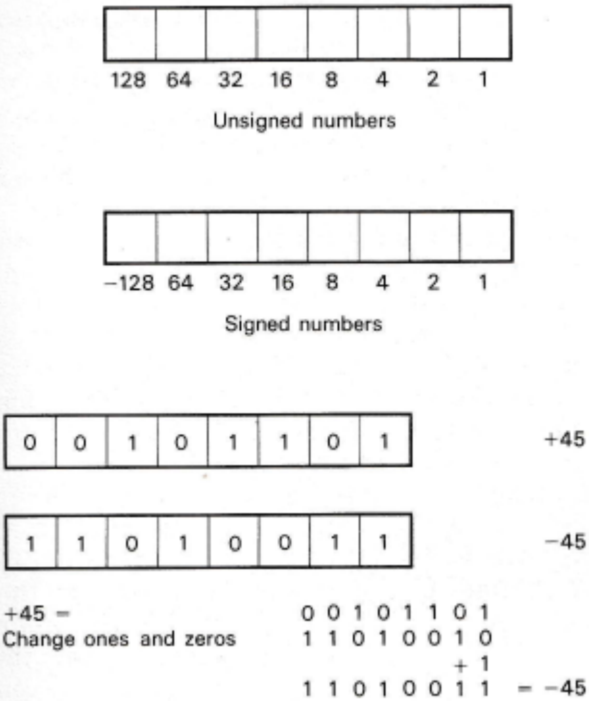

128 64 32 16 8 4 2 1
Unsigned numbers
−128 64 32 16 8 4 2 1
Signed numbers
0 0 1 0 1 1 0 1
+45
1 1 0 1 0 0 1 1
−45
+45 = 0 0 1 0 1 1 0 1
Change ones and zeros 1 1 0 1 0 0 1 0
+ 1
1 1 0 1 0 0 1 1 = −45

Figure 2.5

value −128 and all the other digit values represent their normal position value. Using these values, a positive number will only use the seven positive digit positions and the left-hand digit will always be zero; negative values will be obtained by adding the necessary amount to −128. Figure 2.5 gives some examples of both positive and negative signed numbers.

The computer does not use this method to find 'two's complement numbers'. To find the negative number from the positive number, it uses the simple process of taking the binary representation of the positive number and converting all zeros to ones and all ones to zeros, finally adding one to this new number. This is also shown in Fig. 2.5.

2.4 Memory

The memory of the Spectrum can be thought of as a number of boxes, called locations, which can each hold an eight-bit binary number. Because each location holds eight bits it is convenient to refer to the locations as memory bytes, although strictly this should only be used to refer to the contents of a memory location.

The bytes of the memory are numbered in sequence starting from zero. The number allocated to each byte is referred to as its address in exactly the same fashion as houses in a street are numbered to give an address. They are used, in the same way, to enable any particular byte in the memory to be found.

In assembly language programming, the programmer decides which locations in memory will be used for storing data. Often the programmer may wish to use the data which are stored in a particular memory location. A shorthand way of writing, in assembly language, 'the value of the memory byte whose address is' is to enclose the address in brackets, so that, for example, (23 637) refers not to the value 23 637 but to the value of the memory byte at location 23 637.

Inside the Spectrum there are two different types of memory, called ROM and RAM. ROM, which stands for Read Only Memory, is memory that has a program permanently fixed into it; it cannot be changed by the programmer, although the program, or parts of the program, can often be used by the programmer in other programs. The ROM in the Spectrum uses the memory locations from the address 0 to address 16 383. The remainder of the memory is RAM, which stands for Random Access Memory. This is memory which can have its contents changed by the programmer, although there are some locations which it is not advisable to change. No actual harm can be caused by changing any memory bytes of RAM, although in some instances you may cause the computer to 'crash'. This is not as serious as it sounds; at worst it may mean switching off and on again in order to regain control.

The standard 16K machine has 16 times 1024 RAM memory locations. (The letter K in computer jargon stands for the number 1024.) The RAM occupies locations from address 16 384 to address 32 767. The 48K Spectrum has 48K of RAM memory-occupying locations from address 16 384 to address 65 535. This is the biggest memory that can be used on the Spectrum without using special techniques to give the memory unique addresses.

3 ASSEMBLY LANGUAGE PROGRAMMING

3.1 Assembly language

Before any program can be run on a computer, all of its instructions have to be translated into a series of binary numbers. The central processing unit of the computer, in our case the Z80 microprocessor, only understands program instructions in the form of binary numbers.

The major function of the ROM, in the Spectrum, is the conversion of instructions in BASIC into instructions in binary numbers, known as machine code. Since BASIC was not written specifically for the Z80 microprocessor, the conversion to machine code is relatively inefficient and in computer terms it is very slow.

Although it is not possible to write a program and enter it into the Spectrum directly in binary, it is possible to convert the binary numbers into decimal and use the POKE statement to put the instruction into memory. It is worth noting that one of the actions of the POKE statement is to convert the number from decimal into binary before putting it into memory.

Figure 3.1 shows a short program in this decimal form; as you can see, reading this program gives you no idea as to the function of any of the instructions. Entering programs in this form is very tedious and error prone, and any errors can be very difficult to find. Remember, this is only a very short program so you can imagine the problems which would arise with any long program.

In order to use the full capabilities of the Z80 microprocessor, we need a computer language which is not only closely related to the machine code used by the computer but is also easy to read and understand. The language which meets these requirements is *assembly language.*

The main aim of an assembly language is to make the computer's machine code easily understood. For example, the first instruction in Fig. 3.1, decimal 120, converts to the binary number, 01111000B. This represents the instruction to load the contents of register B into the accumulator. In assembly language this instruction would be represented by

```
LD A,B
```

Address	Contents
23760	50
23761	0
23762	120
23763	33
23764	208
23765	92
23766	70
23767	128
23768	50
23769	209
23770	92
23771	201

Figure 3.1

Take a quick look at Appendix A which gives a complete list of all the instructions recognized by the Z80 microprocessor; do not worry if you do not understand many of them as they will all be explained later. Figure 3.2 shows the program from Fig. 3.1 written in assembly language. You should notice that although the number 120 is the first instruction in the machine code program, its assembly language equivalent, LD A,B, is not the first instruction in the assembly language program. It is preceded by instructions which are there to give information to the program which translates this program from assembly language into machine code.

Assembly languages use small groups of letters to indicate the

```
   10 REM go
   12 REM org 23760
   13 REM !save space for the
numbers
   16 REM Num2;defb 50
   18 REM Result;defb 0
   19 REM !load first number
   20 REM ld a,b
   21 REM !load second number
   22 REM ld hl,Num2
   24 REM ld b,(hl)
   25 REM !carry out the addition
   26 REM add a,b
   27 REM !save the result
   28 REM ld (Result),a
   29 REM !return to BASIC
   30 REM ret
   32 REM finish
```

Figure 3.2

operation to be carried out. The letters are chosen to help the programmer to remember which operation is being used. For example, when loading data into one register from another register the letters used are LD and when adding the letters are ADD. There are many more facilities provided in an assembly language; these will be introduced later.

When writing assembly language instructions the positions in the instruction of any spaces, commas, or brackets is critical. For example, the instruction LD (32500),A would be accepted by an assembler such as the ZX Spectrum Assembler, but LD(32500), A or LD (32500)A would not. If, when assembling a program, you receive an error message a worthwhile first step is to carefully check the format of the instruction.

3.2 An example program

It is now time to look at a complete assembly language program and see how it can be used on the Spectrum. The program shown in Fig. 3.3 is a very simple renumbering program. It will only renumber the initial line numbers; it does not renumber line numbers in GOTO or GOSUB statements. At a later stage you may want to attempt to write a more comprehensive routine. Most assembly language programs used on the Spectrum will be run using an accompanying BASIC program. The accompanying BASIC program for the renumber is shown in Fig. 3.4.

At this stage you will probably not understand most of the instructions in the assembly language program although there may be some that you do understand. Of more interest now is the accompanying BASIC program, which is, as you can see, a very short program. The first line sets the system variable, RAMTOP, to leave some protected memory for the machine code program. The next line is used to load the machine code program into this protected memory.

The next line is the most important line of the program; it is the instruction which causes the machine code program to run. The effect of this instruction is to run the machine code program as a subroutine of the BASIC program. If you do not understand how subroutines work do not worry as everything will be explained in a later section. The final instruction lists the renumbered program to show that it has worked.

3.3 Instructions

Generally all machine code instructions used by the

```
   10 REM go
   20 REM org 23760 32500
   22 REM !Program to renumber a
    BASIC program
   23 REM !
   25 REM !find start of the
program
   30 REM ld hl,23635
   35 REM !set DE to zero
   40 REM ld de,0
   50 REM Loop;ex de,hl
   53 REM !Set increment between
lines
   55 REM ld bc,10
   58 REM !calculate the line
number
   60 REM add hl,bc
   70 REM ex de,hl
   75 REM !store the line number
   80 REM ld (hl),d
   90 REM inc hl
  100 REM ld (hl),e
  110 REM inc hl
  115 REM !find the length of the
line
  120 REM ld c,(hl)
  130 REM inc hl
  140 REM ld b,(hl)
  150 REM dec hl
  160 REM add hl,bc
  165 REM !find number of next
line
  170 REM ld d,(hl)
  180 REM inc hl
  190 REM ld e,(hl)
  200 REM dec hl
  205 REM !test for end of
program by looking for line 9000
  210 REM ld a,d
  220 REM sub 35
  230 REM jp m,Loop
  240 REM ld a,e
  250 REM sub 40
  260 REM jp m,Loop
  270 REM ret
  280 REM finish
```

Figure 3.3

```
9000 CLEAR 32500
9010 LOAD "figure 3.3"CODE
9020 RANDOMIZE USR 32500
9030 LIST
```

Figure 3.4

microprocessor consist of two parts. The first part, called the operation, tells the computer what action to take while the second part, called the operand, tells the computer what data to use. The first instruction in Fig. 3.1 is the decimal number 120, which converts to the binary number 01111000. The first five digits of this number, 01111, represent the code for the operation 'load into the accumulator' and the final three digits, 000, represent the operand—in this case the data have the value in the B register. The instruction in full is 'load into the accumulator the value in the B register'.

A further look at Appendix A will show that the instructions are not of equal length: some use only one byte, some two, some three, and some four bytes. These differences are mainly due to different ways of specifying operands; they are known as addressing modes and will be detailed later.

Figure 3.5 shows several instructions of different lengths in both assembly language and machine code.

Assembly language	Machine code	
LD A,B	01111000B	120
LDIR	11101101B	237
	10110000B	176
JP 32000	11000011B	195
	00000000B	0
	01111101B	125
LD (32000),IX	11011101B	221
	00100010B	34
	00000000B	0
	01111101B	125

Figure 3.5

3.4 Assembling

A program that has been written in assembly language must be translated into machine code before it can be run by the computer. Unlike programs in BASIC which are translated into machine code and then run one line at a time, all of an assembly language program is translated into machine code before any of the program is run. This means that an area of memory has to be used to store the machine code program in addition to the memory used to hold the assembly language program. Figure 3.6 illustrates a typical use of memory with an assembly language program.

The most convenient method of carrying out the translation is to use a program known as an assembler; this will carry out the translation into machine code, check for errors in the program, and load the programs into memory. All of the programs in this book have been assembled using the ZX Spectrum Machine Code Assembler. This program really does extend your Spectrum. Appendix B describes the use of this assembler.

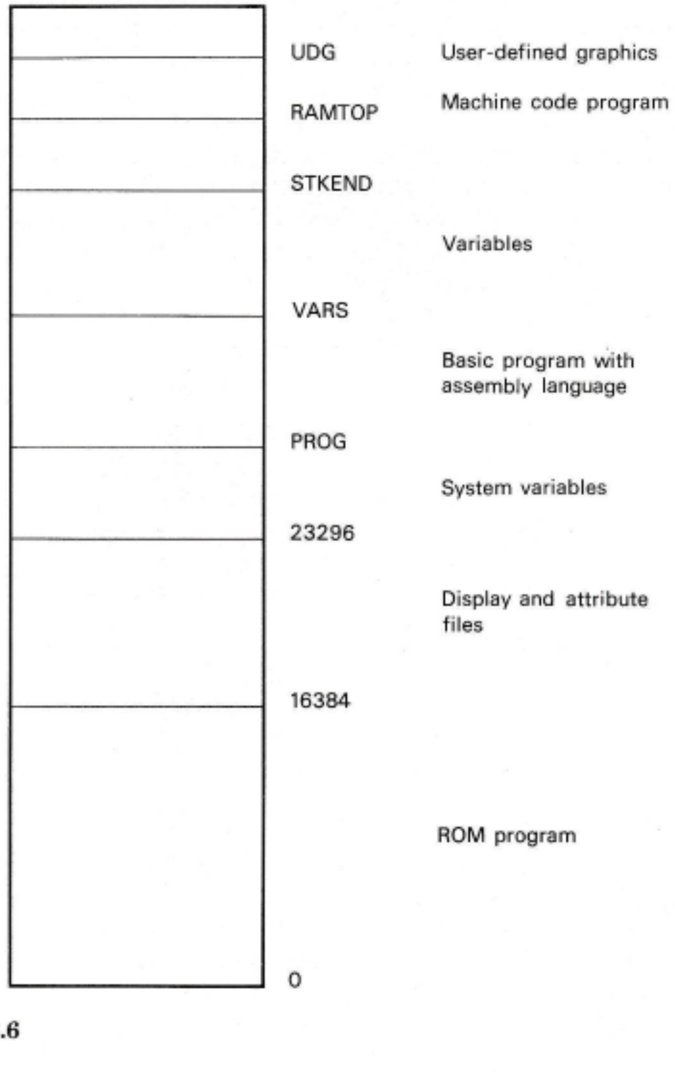

UDG
User-defined graphics
RAMTOP
Machine code program
STKEND
Variables
VARS
Basic program with assembly language
PROG
System variables
23296
Display and attribute files
16384
ROM program
0

Figure 3.6

If an assembler is not available, programs can be translated manually; this process is known as hand assembly and is described in detail in Appendix D.

When the program has been translated into machine code, it can be run by the computer and further errors in the program can be found and corrected. Compared with BASIC programs, this is a more difficult process because machine code programs which go wrong do not produce error messages. Another major problem with machine code programs is that the BREAK key has no effect, unless it is specifically written into the program. A method of doing this will be described later. If a machine code program goes into an indefinite loop—this is usually indicated by the computer appearing to do nothing—the only way to recover control is to switch off the computer and then switch back on again. You will find that it is a good idea to get into the habit of saving your programs before you attempt to run them. Then if you are unfortunate enough to get into an indefinite loop you will not have to reenter all of your program.

Since the Spectrum is designed to run mainly BASIC programs, all of your machine code programs are run as subroutines to the BASIC system. The normal method of starting a machine code program is with the statement or command RANDOMIZE USR XXXX, where XXXX is the starting location in memory of the machine code program. The function USR is a special type of subroutine call; it is used for a machine code subroutine in the same way as GOSUB is used for a basic subroutine. As all machine code programs are run as subroutines to the BASIC system they must all end with the instruction RET, which means return from a subroutine, so that control is passed back to the BASIC system.

3.5 Program storage

The assembly language programmer works much closer to the microprocessor in the computer than the programmer using BASIC. This gives the programmer much greater control over and use of the facilities provided in the microprocessor; however, it also means that he or she can take less for granted and a number of tasks which are carried out automatically in BASIC must be performed by the programmer. One of these tasks is to decide where, in the computer's memory, the machine code program should be stored so that it can be run easily.

There are essentially three different areas of memory which can be used, all with their own advantages and disadvantages.

The safest place to store your programs is at the top of memory. The program can be loaded into the top of memory and the system variable RAMTOP set to the memory location just below the start of the program; the program is then protected from overwriting by BASIC. The main disadvantage of this part of memory is that the machine code program and its accompanying BASIC program, if any, must be saved and loaded separately.

If you have used machine code on a ZX81 computer you will probably have stored your programs in a REM statement at the start of a BASIC program. This can still be done on the Spectrum and it has the advantage that the machine code program is automatically saved with the BASIC program. The disadvantage of this method is that the start of a BASIC program is not a fixed position on the Spectrum but depends on the devices attached to the Spectrum. The start of the program area is found from the system variable PROG.

Finally, the program can be stored in the spare area of memory between the calculator stack and the machine stack. This area is pointed to by the system variable STKEND.

Probably the best method of working is to use a REM statement for storing your program while you are working on it and getting it to work, but when the program is working it should be stored at the top of memory and protected from BASIC by resetting the RAMTOP variable using the CLEAR statement.

Figure 3.7 shows these three areas of memory.

3.6 Subroutines

Subroutines are a very important programming technique, especially for the assembly language programmer. Because of their importance this section will show why they are used and how to use them. It will not look at the details of how they work; this will be covered in a later chapter.

Why should you use subroutines? Suppose you write a program containing two or more groups of statements which carry out the same action. It is wasteful of time and memory to keep repeating the same set of statements within the program; instead you write the set of statements once, as a block on their own. This separate block of statements is a subroutine. Usually subroutines are written separately after the main program. When the program is run, every time the subroutine is needed a special jump instruction is used to start the subroutine running. At the end of the subroutine another special jump instruction takes the

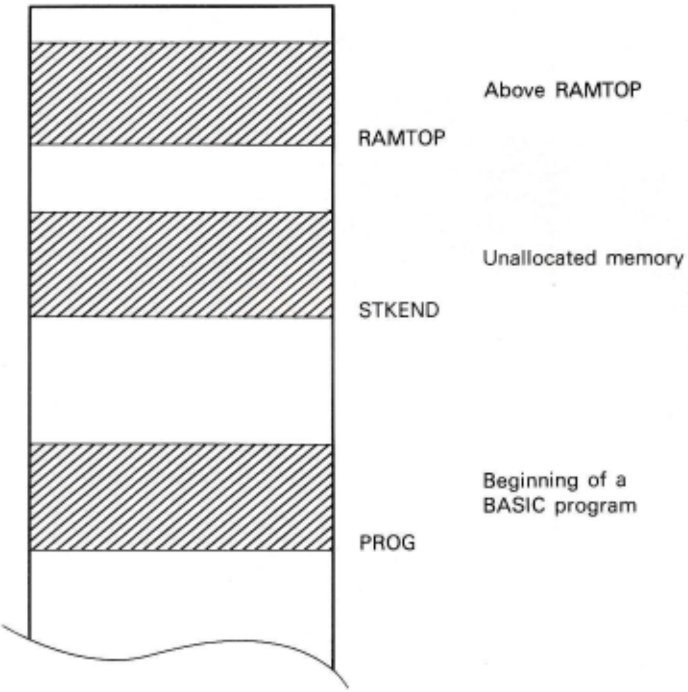


Figure 3.7

computer back to the instruction in the main program following the instruction which caused the jump to the subroutine.

The assembly language for the Spectrum uses the instruction CALL to cause a jump to a subroutine. CALL must be followed by the location in memory of the first memory byte of the subroutine. As you will see later, it is possible to give important memory bytes names, or *labels;* the label of the memory byte can then be used in place of the numeric address.

Figure 3.8 is an example of a short program which uses a subroutine to multiply a list of numbers by 10. Figure 3.9 is a short BASIC program which uses the program in Fig. 3.8. At this stage, the main details of the program are irrelevant; you should only note that the subroutine is caused to run by the instruction:

```
CALL MULTEN
```

where MULTEN is a label which indicates where the subroutine starts. When the program is translated into machine code, the label will be converted into the number of the memory location which holds the first instruction of the subroutine.

The final instruction in any subroutine is RET. This causes the

```
  10 REM go
  20 REM org 23766
  30 REM !use of a subroutine
  32 REM !load 1st number
  35 REM ld a,(23760)
  38 REM !jump to subroutine
  40 REM call Multen
  43 REM !save the result
  45 REM ld (23763),a
  48 REM !repeat for 2nd number
  50 REM ld a,(23761)
  55 REM call Multen
  60 REM ld (23764),a
  62 REM !repeat for 3rd number
  65 REM ld a,(23762)
  70 REM call Multen
  75 REM ld (23765),a
  80 REM ret;!end of main progra
m
  85 REM !start of subroutine
  90 REM Multen;add a,a;!times 2
  95 REM ld b,a;!temp storage
 100 REM add a,a;!times 4
 105 REM add a,a;!times 8
 110 REM add a,b;!8 times+2 time
s
 115 REM ret;!return to main pro
gram
 120 REM finish
```

Figure 3.8

```
   1 REM 00000000000000000000000
00000000000000000000000000000000
0000000000000000000000000000
  10 INPUT a,b,c
  20 LET x=23760
  30 POKE (x+0),a
  40 POKE (x+1),b
  50 POKE (x+2),c
  60 RANDOMIZE USR 23766
  70 FOR i=3 TO 5
  80 PRINT PEEK (x+i)
  90 NEXT i
```

Figure 3.9

computer to jump back, or return, to the correct place in the main program. All machine code programs on the Spectrum are run with the USR function; as this is a jump to a subroutine all machine code programs must end with a RET instruction.

A program may use any number of subroutines as long as the start of each subroutine is clearly indicated and each subroutine ends with a RET instruction. A program with more than one subroutine is usually easier to understand if all the subroutines are located at the end of the main program, one after the other.

One of the advantages of subroutines is that you can use subroutines written by other people without needing to know the details of the subroutine. A useful source of subroutines for the Spectrum are the routines in the Sinclair ROM. Appendix G lists some of these routines and how they may be used; this is not a full list of the useful routines but just some of the easiest to use.

4 SOME SIMPLE INSTRUCTIONS

4.1 Data in the computer

In this chapter we shall look at the assembly language instructions which allow bytes of data to be moved about inside the computer and simple arithmetic to be carried out with numbers stored in one byte. All the data which are used in an assembly language program must be moved into their correct storage locations by the programmer. Much of an assembly language program consists of instructions for moving data to appropriate registers or memory locations.

Simple arithmetic instructions account for another large percentage of a program. The most-used arithmetic instructions are those which allow the value in a storage location, either a register or a memory byte, to be increased or decreased by one.

4.2 Loading registers

Data which are being processed or will be required shortly for processing are stored in the registers of the central processing unit. Data which are held in a register can be accessed in about half the time taken to move data from the main memory. Of course, only a very limited amount of data can be held in the registers

A numeric value can be put directly into any of the single, eight-bit, registers using the instruction with the form:

```
LD r,n
```

where n is the value to be put into the register and r is any of the eight-bit registers A, B, C, D, E, H, or L. For example, the instruction:

```
LD D,20
```

will put the binary equivalent to the number 20 as an eight-bit number into the D register. If you could see the contents of the D register it would look like Fig. 4.1.

0	0	0	1	0	1	0	0

Figure 4.1

Normally the value in a register will be treated as a positive whole number and it must be within the range which can be held in eight bits; it must be between 0 and 255. The computer can also be programmed to treat the value in a register as a signed, or two's complement, number. The range of numbers which can be held is between −128 and +127. Signed numbers were described in Chapter 2.

The A register, which is also called the accumulator, is the most important register for the programmer. All logical operations and most eight-bit arithmetic operations involve the use of the value in the accumulator and normally the result of the processing is left in the accumulator.

Because the registers are often used as general purpose storage locations for data awaiting processing, there is a need to move data between registers. The instruction which allows data to be moved between registers is:

LD rl,r2

The effect of this instruction is to put a copy of the value in register r2 into register rl. The value in register r2 is unchanged by the instruction. The registers, rl and r2, can be any of the eight-bit registers A, B, C, D, E, H, or L. For example, if the A register contains the value 47 and the C register the value 127, the instruction LD C,A will put the value 47 into the C register and the value 47 will be left in the A register. Figure 4.2 shows the contents of the registers before and after the instruction.

'A' register								'C' register							
0	0	1	0	1	1	1	1	0	1	1	1	1	1	1	1

LD C,A

0	0	1	0	1	1	1	1	0	0	1	0	1	1	1	1

Figure 4.2

4.3 Increment and decrement

The most common arithmetic operations required by the programmer are to increase or decrease the value in a register or memory location by one. To carry out these operations there are two instructions with the form:

INC r and DEC r

These instructions have the effect of increasing by one or decreasing by one the value in one of the eight-bit registers A, B, C, D, E, H, or L, or one of the 16-bit register pairs BC, DE, or HL. These instructions can also be used to increase or decrease the value in a memory location by first putting the address of the memory location into the HL register pair and then using the HL register pair as a pointer. For example, if the HL register pair contains the value 32 000 and memory location 32 000 contains the value 58 then the instruction INC (HL) will increase the value in memory location 32 000 by one to 59. Remember that the brackets are a shorthand way of referring to the value in a memory location. Figure 4.3 illustrates this use of the HL register pair as a pointer.

4.4 Memory transfers

Most of the data used by the programmer will be stored in memory and will need to be transferred to and from the registers in the central processor before and after processing. The accumulator, or A register, is the register most likely to be used for the processing, and it is the register which has the greatest flexibility when moving data to, and from, memory.

All instructions which use data in memory contain the address of the memory location or they use a register pair as a pointer to the memory location. The simplest instruction for moving data to the A register is:

LD A,(nn)

where nn is the address of the memory location which contains the data. Remember that although a memory location can only contain one byte, i.e., eight bits of data, the address of a memory location requires a 16-bit binary number. So nn is a number in the range 0 to 65 535.

Similarly, data can be moved from the A register to memory using the instruction:

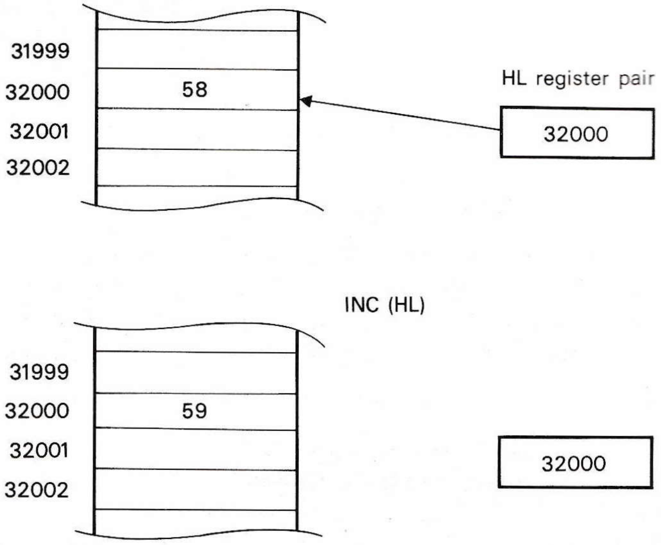


Figure 4.3

LD (nn),A

It is unusual to be able to specify directly in an instruction the address of the memory location. Often the address will have been calculated, by a previous part of the program, and will be held in one of the 16-bit register pairs. The instructions with the form:

LD A,(rr) and LD (rr),A

where rr is one of the BC, DE, or HL register pairs, use the value in the 16-bit register as the address of the memory location which is involved in the move. As stated for the earlier MOVE instructions, the effect of these instructions is to copy the data from memory to the accumulator, or vice versa, without changing the value in the source of the data.

It was stated earlier that most of the secondary registers in the central processing unit (CPU) tend to be used for specific purposes and the first of these we shall look at is the HL register pair. This forms a 16-bit register whose main purpose is to point to locations in memory; in other words, it is the register normally used to hold the address of data to be moved to or from memory. Using the instructions:

LD r,(HL) and LD (HL),r

data can be moved between any of the eight-bit registers and memory. Again using the HL register pair as a memory pointer, a memory location can be loaded directly with a numeric value. The form of the instruction is

```
LD (HL),n
```

where n must be a number between 0 and 255.

4.5 Addition and subtraction

We can now look at some very simple arithmetic. The Z80 microprocessor in the Spectrum has instructions which let us add to or subtract from the value in the accumulator. The instructions to carry out these operations using a numeric value directly are:

```
ADD A,n    and    SUB n
```

where n is a number between 0 and 255.

You should have noticed that the form of these two instructions is different. This is because, although the A register is the only eight-bit register which can be used with the ADD instruction, 16-bit addition can be carried out using the HL register pair, but subtraction using the SUB instruction can only be carried out using eight-bit values and the accumulator. This means that in the ADD instruction the use of the A register must be shown in the instruction, but this is unnecessary with the SUB instruction because the A register is always used. I must stress that the SUB instruction would be wrong if it is written in the same form as the ADD instruction; it is not optional to leave out the reference to the A register.

When addition or subtraction are carried out using the accumulator the result of the calculation is left in the accumulator. For example, the effect of the instructions:

```
LD A,52
SUB 19
ADD A,22
INC A
```

is to put the value 52 into the accumulator, subtract 19 from this leaving the value 33 in the accumulator, add 22 to this to give the value 55, and finally increment this by one to leave the value 56 in the accumulator.

The value in any of the eight-bit registers, including the accumulator, can be added to, or subtracted from, the value in the accumulator with the instructions:

```
ADD A,r    and    SUB r
```

Using HL as a memory pointer, as shown earlier in the chapter, a value in memory can also be added to, or subtracted from, the value in the accumulator.

```
  10 REM go
  20 REM org 23760
  25 REM !load location of first
data item
  30 REM ld hl,32500
  35 REM !load first number
  40 REM ld a,(hl)
  50 REM inc hl
  55 REM !load second number
  60 REM ld b,(hl)
  65 REM !add them together
  70 REM add a,b
  80 REM ld b,a
  85 REM !double the result
  90 REM add a,a
  95 REM !store the results
 100 REM ld (hl),b
 110 REM dec hl
 120 REM ld (hl),a
 130 REM finish
```

Figure 4.4

Figure 4.4 is a short segment of program which illustrates some of the instructions shown in this chapter. The first instruction puts the value 32 000 into the HL register pair. The value in memory location 32 000 is then copied to the A register using HL as a memory pointer. The next instruction increases the value in HL by one to 32 001 and then the value in this memory location is copied into the B register. Now the two values are added and the result is copied from the accumulator into the B register. Next the value in the accumulator is doubled, by adding it to itself, and finally the values in the A and B registers are copied back into memory.

If at the start of the program memory location 32 000 contains the value 20 and memory location 32 001 contains the value 15, what would be the values in these two memory locations at the end of the program? If you have any problems with this question Fig. 4.5 shows the values in the registers and memory locations after each of the instructions in the program segment.

4.6 Writing a program

The program shown in Fig. 4.6 is a useful subroutine for the writer of games programs. It is a program which will cause a line

Instruction	HL	A	B	Location 32500	Location 32501
LD HL,32500	32500	?	?	20	15
LD A,(HL)	32500	20	?	20	15
INC HL	32501	20	?	20	15
LD B,(HL)	32501	20	15	20	15
ADD A,B	32501	35	15	20	15
LD B,A	32501	35	35	20	15
ADD, A,A	32501	70	35	20	15
LD (HL),B	32501	70	35	20	35
DEC HL	32500	70	35	20	35
LD (HL),A	32500	70	35	70	35

Figure 4.5

```
  10 REM go
  20 REM org 23760
  22 REM Temp;defb 0
  25 REM !program to scroll top
line sideways
  30 REM ld hl,16384;!start of
 line
  40 REM ld c,8;!no. of rows/lin
e
  50 REM Loopa
  55 REM ld a,(hl)
  60 REM ld (Temp),a;!save first
 character
  70 REM ld b,31;!no. of charact
ers to move
  75 REM !loop to move parts of
 characters
  80 REM Loopb
  85 REM inc hl
  90 REM ld a,(hl)
 100 REM dec hl
 110 REM ld (hl),a
 120 REM inc hl
 130 REM djnz Loopb
 140 REM inc hl
 145 REM !put 1st at end of line
 150 REM ld a,(Temp)
 160 REM ld (hl),a
 165 REM !find next part of line
 170 REM ld de,225
 180 REM add hl,de
 190 REM dec c
 200 REM jp nz,Loopa
 210 REM ret
 220 REM finish
```

Figure 4.6

of characters on the screen to scroll to the left. It is an appropriate program to look at now because much of the program consists of data movement and simple arithmetic.

The first instruction is a directive, an instruction to the assembler program to set aside one location in memory. Now we start the program proper when we load the HL register pair with the location in memory of the first byte of data for the line to be scrolled. In the example shown it is the top line. Using the data given in a later chapter about the display file, you will be able to modify this program to scroll any line. To understand the program you need to know that each line of characters on the screen is stored in memory as eight blocks of 32 bytes and there is a 224-byte block between each of these blocks.

If you look at the program now you should be able to understand all of the instructions except for DJNZ Loopb and JP NZ, Loopa. JP NZ means 'jump if the previous arithmetic instruction gave a non-zero result' and DJNZ is a combination of the instructions DEC B and JP NZ.

4.7 Labels

When you are programming in BASIC you will often use instructions which refer to other instructions in the program (e.g., when using GOTO instructions). In BASIC, every line of the program is given a line number and this number is used when referring to the line from other instructions. Machine code programs do not have any line numbers when they are stored in memory. Program instructions which refer to other instructions in the program do so by giving the address of the memory location which holds the first byte of the instruction.

Calculating the memory location which holds the first byte of a particular instruction is very difficult and time-consuming. In assembly language this problem is overcome by using labels, which are names given to particular instructions in the program. When the instruction is referred to by another instruction, its label is used. When the program is assembled into a machine code program, the assembler program automatically converts the labels into the correct memory address.

Unlike BASIC, only instructions which are referred to by other instructions are given labels. The label is chosen by the programmer, within the rules set by the assembler programmer.

4.8 Program

At this stage the programs we can write are very limited and must be simple subroutines which can be called from a BASIC program. Write an assembly language subroutine which will multiply two numbers together by adding the first number to itself the number of times given by the second number. For example, 4×3 is the same as 4+4+4. To write this program you will need the instruction JP NZ, label.

Assemble the program into the top of memory and reset RAMTOP to leave three protected memory locations before the start of your program. For example, on the 16K Spectrum set RAMTOP to 32 499 and assemble the program to start at location 32 503 using 'org 32503'. The two numbers to be multiplied together will be in memory locations 32 500 and 32 501 and the result should be put in memory location 32 502.

Write a BASIC program which will input two numbers and POKE them into locations 32 500 and 32 501. Then call the machine code subroutine and finally print the result by peeking location 32 502.

You may have noticed from an earlier program that the HL register pair can be loaded directly with a number (see Fig. 4.4).

5 JUMPING AROUND

5.1 Why jump?

One of the things you will know from your BASIC programming is that very few programs start at the first instruction and go through to the last instruction carrying out each instruction once only and in the order of their line numbers. All but the very simplest of programs will contain jumps, which change the order in which the instructions are carried out. There are two types of jump instruction—unconditional and conditional.

5.2 Unconditional jumps

In BASIC the unconditional jump instruction is GOTO line number; in assembly language the instruction is:

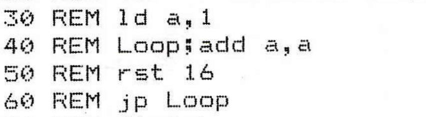
JP nn

where nn is the address of a memory location. The effect of the instruction is to take the address nn as the first byte of the next instruction to be run. Normally in assembly language the number nn will be replaced by a label. For example, in a program assembled to start at location 32 500 in memory and with the first instruction given the label 'Start', the instructions:

JP 32500   and   JP Start

are equivalent.

Unconditional jumps on their own are not very useful because they will usually produce indefinite loops, as shown in Fig. 5.1.



```
10 REM go
20 REM org 23760
25 REM !an indefinite loop
30 REM ld a,1
40 REM Loop;add a,a
50 REM rst 16
60 REM jp Loop
70 REM finish
```

Figure 5.1

Unfortunately, on the Spectrum, if the program is in an indefinite loop, the only way to stop the program is to switch off the computer and start again.

The JP instruction allows the computer to jump to its next instruction anywhere within the available memory of the computer; in most programs, the majority of jumps are made to instructions within a few bytes of the jump instruction. To take advantage of this there is a second unconditional jump instruction which allows these short jumps to be made. This is the relative jump instruction, which only occupies two bytes of memory compared to the three bytes used by the JP instruction. It is called a relative jump because the data part of the instruction specifies the number of memory locations from the end of the jump instruction to the beginning of the instruction which is to follow the jump.

The form of the instruction is:

JR n

where n is a number in the range −128 to +127. Again the instruction to be executed after the jump can be referred to by a label, using the instruction, JR label. When the program is assembled, the assembler will calculate the number of bytes to be jumped and replace the label with the correct relative jump. When using labels, you must be sure that the required jump will be within the range of the instruction; otherwise an error will occur when the program is assembled. if you have any doubt you should use the JP instruction.

5.3 The flag register

Among the registers in the Z80 microprocessor there is an eight-bit register which we have not yet discussed; this is the F register. This register is not used to hold data but is a group of separate bits, most of which are used to hold information concerning the result produced by arithmetic or logic instructions. Figure 5.2 shows the names which are given to the various bits.

Because they are used to indicate the result from a previous instruction they are known as flags. Two important bits of the F register are the *sign flag* and the *zero flag*.

If an arithmetic or logic operation produces a negative result the sign flag is set to one, while a zero or positive result sets the flag to zero. If an instruction produces a zero result then the zero flag is set to one; any non-zero result, either positive or negative,

S	Z	X	H	X	P/V	N	C

	Flag		Test			
S	Sign	P	Positive	or	M	Minus
Z	Zero	Z	Zero	or	NZ	Not zero
H	Half-carry	–				
P/V	Parity/overflow	PO	Parity odd	or	PE	Parity even
N	Add/subtract	–				
C	Carry	C	Carry	or	NC	No carry
X	These bits not used					

Figure 5.2

sets the flag to zero. We shall look at the rest of the flags later in the book.

Figure 5.3 shows a program segment with the value of the sign and zero flags after the execution of each instruction.

	Flags	
Program	Sign	Zero
LD A,10	?	?
SUB 11	1	0
LD B,A	1	0
INC B	0	1
INC B	0	0

Figure 5.3

5.4 Conditional jumps

You will know from programming in BASIC that unconditional jumps have a very limited use on their own; they are almost always used with instructions which make a jump dependent upon the results of a test. This kind of jump is known as a conditional jump.

When a program reaches a conditional jump instruction it either continues with the instruction after the jump or it jumps to an instruction elsewhere in the program. The jump only takes place if the required condition is met; in assembly language the condition is indicated by one of the bits in the flag register. Figure 5.4 shows a short program which uses a conditional jump instruction to jump out of a short loop. The program adds together numbers in successive memory locations until it reaches a

```
  10 REM go
  20 REM org 23760
  30 REM equ 32550 Start
  40 REM ld hl,Start
  50 REM ! Start of the loop
  60 REM Loop;ld a,(hl)
  70 REM ! Check whether it is z
ero
  75 REM sub 0
  80 REM jr z,Endit
  90 REM add a,b
 100 REM ld b,a
 105 REM inc hl
 110 REM jr Loop
 120 REM Endit;ld a,b
 125 REM ld hl,Start
 127 REM !save the result
 130 REM ld (hl),a
 135 REM ret
 140 REM finish
```

Figure 5.4

location which holds a zero value. This location is then used to hold the sum.

The form of the conditional jump instruction is:

JP c,addr

where c is the condition to be tested and addr is the memory location which holds the instruction to be executed if the condition is met. The address will, of course, normally be replaced by a label in an assembly program.

Some of the conditions which can be tested are:

Zero	JP Z, LABEL1
Non-zero	JP NZ,LABEL2
Minus	JP M,LABEL3
Positive	JP P,LABEL4

These conditions are tested by the computer inspecting the value of the relevant bit of the flag register.

In BASIC, jumps can be made on any arithmetic condition, but in assembly language these must be rewritten as tests for sign or zero results. Any arithmetic condition can be written in this form. Figure 5.5 shows a short section of a BASIC program and its equivalent in assembly program. Using subtraction and a test for sign or zero any of the main conditions used in BASIC can be

```
BASIC                        Assembly

10 LET A=10                           LD A,10
20 PRINT A                   LOOP;    RST 16
30 LET A=A+10                         ADD A,10
40 IF A>100 THEN 60                   SUB 100
50 GO TO 20                           JP P,END
60 STOP                               JP LOOP
                             END;     RET
```

Figure 5.5

programmed. Figure 5.6 shows the assembly language equivalent of the four main conditions. Using a combination of tests, any test which can be programmed in BASIC can be carried out in assembly language.

We can also carry out relative conditional jumps. The conditions which can be tested by a relative jump are limited, so a relative jump can examine the zero flag but not the sign flag. The instructions using the zero flag are:

JR Z,disp   and   JR NZ,disp

where disp is the number of bytes of the instruction which is to be executed if the condition is met. Again this will normally be replaced by a label and the number of bytes will be calculated by the assembler.

5.5 Comparisons

There is a problem using subtraction to set the condition flags before a jump instruction, because the original value in the accumulator is changed by the subtraction. It is often possible to restore the value by an addition, but this can be difficult and

```
BASIC                  Assembly

IF A=B THEN            SUB B
                       JP Z,NEXT

IF A>B THEN            SUB B
                       JP P,NEXT

IF A<B THEN            SUB B
                       JP M,NEXT

IF A<>B THEN           SUB B
                       JP NZ,NEXT
```

Figure 5.6

inconvenient. The Z80 microprocessor in the Spectrum has an instruction which overcomes this problem. It is called a compare instruction and has the form:

```
CP n
```

where n is an eight-bit value, or the value in any of the eight bit registers, or a value in memory whose address is held in the HL register pair. The instruction allows the value in the accumulator to be compared with another eight-bit value without changing the value in the accumulator. The instruction subtracts the specified value from the value in the accumulator and sets the bits in the flag register according to the result. The result of the subtraction is then discarded and the original value remains in the accumulator.

The main use of the compare instruction is to determine whether the value in the accumulator is a specific value; often this will be used after an input from the keyboard to check whether a particular character has been input. After a keyboard input the accumulator will hold the code for the character. For example, the instructions:

```
CP 65
JR Z,ALABEL
CP 13
JP NZ,LOOP
```

following a keyboard input will first check whether the character was an 'A' and if so jump to the instruction labelled ALABEL. If it was not an 'A' it would then check to see if the 'ENTER' key was pressed and it would jump to the instruction labelled LOOP if it was not pressed.

Not all instructions set the bits in the flag register, e.g., the instruction LD A,(HL) will not set the flag register. It is often useful to have the bits in the flag register set according to the value in the accumulator, even when the instruction does not effect the flag register. The instruction CP 0 can be used to set the flag register by comparing the accumulator with zero. Appendix A summarizes the instructions which do affect the flag register and the bits which are affected. None of the data movement instructions we have met so far affect the flag register; if we wish to test the sign of a value copied from memory we could use the CP 0 instruction, as shown in the following program segment:

```
LD HL,32000
NEXT: INC HL
```

```
LD A,(HL)
CP 0
JP M,NEXT
```

This program will loop until it finds a positive value in memory.

5.6 Pseudo-operations

Pseudo-operations are instructions in assembly language programs which do not produce any equivalent machine code instructions. The two main uses of pseudo-operations are to pass information to the assembler program regarding how the program is to be translated and to tell the assembler program to allocate space, in memory, for data.

Some pseudo-operations are only used with one assembler program but others will be understood by most assemblers. The programs in this book were prepared using the ZX Spectrum Machine Code Assembler. This assembler uses instructions which are written in REM statements in a BASIC program and the first instruction in any program must be the pseudo-operation:

```
GO
```

and the last instruction in any program must be the pseudo-operation:

```
FINISH
```

These instructions are only recognized by this assembler; another pseudo-operation which must be used with this assembler and most other assemblers for the Spectrum is:

```
ORG nn
```

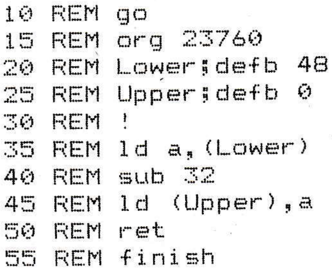
This instruction tells the assembler program that the next instruction should be loaded into memory location nn. Most assemblers require this instruction at the start of the program to tell the assembler where to start the program in memory. It can also be used in the middle of a program to change the location for the next instruction.

The ZX Spectrum Machine Code Assembler allows another very useful form of the ORG statement; this is:

```
ORG a b
```

where a is the starting location in memory to which the program is loaded by the assembler, but the program is translated as if its starting address is location b. On the Spectrum, the best place for a machine code program is at the top of memory, but when you are producing and assembling assembly language programs, the top of memory is occupied by the assembler program. This form of the instruction allows code to be translated into low memory and transferred to high memory for running.

The most useful of the pseudo-operations is the instruction DEFB. This instruction tells the assembler program to reserve the following memory locations to store eight-bit values and it is followed by the values to be put in these locations. When the instruction is followed by more than one value they are separated by spaces. Usually the instruction is preceded by a label and the first location can then be referred to by name. Figure 5.7 is a short program which shows the use of the instruction to allocate two memory locations. The effect of the program is to subtract 32 from the number in location LOWER and put the result in location UPPER. These names were chosen because this short program can be used to change small (lower case) letters into capital (upper case) letters.



```
10 REM go
15 REM org 23760
20 REM Lower;defb 48
25 REM Upper;defb 0
30 REM !
35 REM ld a,(Lower)
40 REM sub 32
45 REM ld (Upper),a
50 REM ret
55 REM finish
```

Figure 5.7

5.7 Screen output

The Spectrum, like most microcomputers, uses a technique called memory mapping to produce its display on a television or monitor screen. Memory mapping consists of allocating an area of the computer's memory to hold all the details which are shown on the display screen. Part of the monitor program is used to activate some of the computer's circuits to copy the display from memory to the television output and hence to the screen.

One way of producing a display on the screen is to load the details of the display into memory and it will then be automatically displayed on the screen. The Spectrum actually allocates two areas of memory to the screen display. One area is used to hold the shapes to be displayed and the second area holds the colour details for the display. Because the Spectrum is capable of producing high-resolution graphic displays the display which is held in memory for the shape of any character is contained in eight separate memory locations. This makes it more difficult to put characters on the screen by loading their details into memory.

However, because the display of characters is a common requirement there is a subroutine in the Sinclair ROM which carries out this function. This routine can be called using the instruction:

```
RST 16
```

Put simply, the effect of this instruction is to use the value in the accumulator as a character code and to display the character on the screen. The RST 16 routine is a very powrful routine because, as well as displaying a character giving its character code, it will also recognize and act upon the print control characters.

To control input and output, the Spectrum system uses a technique known as channelling, which uses a separate part of the ROM program for each of its different forms of input and output. The standard Spectrum system uses four channels:

Channel 1—allows input from the keyboard and output to the lower part of the screen.

Channel 2—allows output to the upper part of the screen, but no input.

Channel 3—allows use of the BASIC workspace; this is unlikely to be useful.

Channel 4—allows output to the printer, but no input.

When the computer is switched on it initializes to channel 1, waiting for an input from the keyboard. To use the other channels, the channel number is loaded into the accumulator and a ROM subroutine is called to activate that channel. Figure 5.9 shows the use of this routine to give output to the upper part of the screen. Appendix E shows the ASCII codes for the characters which can be input from the keyboard and then printed and Appendix F shows the control characters recognized by the RST 16 routine.

Figures 5.8 and 5.9 are programs using RST 16 to print in the top and lower halves of the screen. Figure 5.8 is a very simple program, but Figure 5.9 is more complicated and shows how the RST 16 routine can be used to produce an animated display. At

```
   10 REM go
   15 REM org 23760
   17 REM !do it 10 times
   20 REM ld b,10
   25 REM Loopa;ld c,5
   27 REM !5 spaces at start of
 line
   30 REM Loopb;ld a,32
   35 REM rst 16
   40 REM dec c
   45 REM jr nz,Loopb
   47 REM !load the codes for the
  letters
   50 REM ld a,77;!  m
   55 REM rst 16
   60 REM ld a,101;! e
   65 REM rst 16
   70 REM ld a,115;! s
   75 REM rst 16
   77 REM ld a,115;! s
   80 REM rst 16
   85 REM ld a,97;!  a
   90 REM rst 16
   95 REM ld a,103;! g
  100 REM rst 16
  105 REM ld a,101;! e
  110 REM rst 16
  113 REM !go to next line
  115 REM ld a,13;! ENTER
  120 REM rst 16
  125 REM dec b
  130 REM jr nz,Loopa
  135 REM ret
  140 REM finish
```

Figure 5.8

```
   10 REM go
   20 REM org 23760
   30 REM ! Animated display
   40 REM ! Initialise
   50 REM ld d,10; ! line no
   60 REM ld e,15; ! col no
   90 REM ld b,127;!moving charac
ter
  100 REM call Print
  110 REM ! Up and right
  120 REM Ur;ld h,d
  130 REM ld l,e; ! Save previous
 position
 140 REM Ur1;dec d; ! next line
 up
 142 REM ld a,d
 145 REM !top of screen
 147 REM cp 1
 150 REM jr z,Dr1
 160 REM Ur2;inc e; ! next col r
ight
 170 REM ld a,e
 180 REM cp 31
 185 REM !side of screen
 190 REM jr z,Ul2
 195 REM !print over old and pri
nt new
 200 REM call Printer
 210 REM jr Ur
 220 REM ! Up and left
 230 REM Ul;ld h,d
 240 REM ld l,e; ! save previous
 position
 250 REM Ul1;dec d; ! next line
up
 252 REM ld a,d
 255 REM !top of screen
 257 REM cp 1
 260 REM jr z,Dl1
 270 REM Ul2;dec e; ! next col l
eft
 272 REM ld a,e
 275 REM !side of screen
 277 REM cp 1
 280 REM jr z,Ur2
 290 REM call Printer
 300 REM jr Ul
 310 REM ! down and right
 320 REM Dr;ld h,d; ! Save previ
ous position
 330 REM ld l,e
 340 REM Dr1;inc d; ! next line
down
 350 REM ld a,d
 360 REM cp 21
 365 REM !bottom of screen
 370 REM jr z,Ur1
 380 REM Dr2;inc e; ! next col r
ight
 390 REM ld a,e
 400 REM cp 31
 405 REM !side of screen
 410 REM jr z,Dl2
 420 REM call Printer
 430 REM jr Dr
 440 REM ! down and left
 450 REM Dl;ld h,d
 460 REM ld l,e; ! save previous
 position
 470 REM Dl1;inc d; ! next line
down
 480 REM ld a,d
 490 REM cp 21
 495 REM !bottom of screen
 500 REM jr z,Ul1
 510 REM Dl2;dec e; ! next col l
eft
 512 REM ld a,e
 515 REM !side of screen
 517 REM cp 1
 520 REM jr z,Dr2
 530 REM call Printer
 540 REM jr Dl
 550 REM Print;ld a,2
 554 REM call 5633;!open channel
 557 REM ld a,22
 560 REM rst 16
 570 REM ld a,d
 580 REM rst 16
 590 REM ld a,e
 600 REM rst 16
 610 REM ld a,b
 620 REM rst 16
 630 REM ret
 640 REM Printer;ex de,hl
 650 REM ld b,32
 660 REM call Print
 670 REM ex de,hl
 680 REM ld b,127
 690 REM call Print
 700 REM ret
 710 REM finish
```

Figure 5.9

this stage this instruction will give us sufficient flexibility for most displays. The production of high-resolution displays will be discussed later.

5.8 Program

Write a program which will display a 6×6 square of '*' characters in the centre of the screen. Figure 5.10 shows the required output.

```
* * * * * *
* * * * * *
* * * * * *
* * * * * *
* * * * * *
* * * * * *
```

Figure 5.10

Use the print control characters to print in the correct position on the screen. Use loops to shorten your program.

As a more difficult example you could write a program to produce the output shown in Fig. 5.11.

```
      *
    * * *
  * * * * *
* * * * * * *
  * * * * *
    * * *
      *
```

Figure 5.11

6 USING THE KEYBOARD

6.1 Keyboard input

Many of our programs require data to be put into the computer from the keyboard. It is a rather long and complex program which is used to enable the computer to understand when and what key has been pressed on the keyboard, but fortunately we can use the subroutines from the Sinclair ROM to do all the hard work.

The main subroutine starts at memory location 703 and it can be used by the instruction:

CALL 703

The effect of the subroutine is to scan the keyboard until a key is pressed and load the character code for the key into the system variable LAST-K, which is at memory location 23 560. This memory location will contain the character code of the last key to be pressed until either a new key press is registered or a different value is loaded into the location.

Figure 6.1 is a short routine which will input a character from the keyboard when a key is pressed, load the code for the character into the accumulator, and then output the character to the screen. This is the normal action of BASIC. You can if you

```
  10 REM go
  20 REM org 23760
1000 REM Cinout;call 4264;!ROM r
outine
1010 REM cp 208;!check key press
ed
1020 REM jr z,Cinout;!none press
ed
1030 REM push af;!save character
 while printing
1040 REM rst 16;!Echo character
1050 REM pop af;!retrieve charac
ter
1060 REM ret;!end of routine
1070 REM finish
```

Figure 6.1

wish input a character from the keyboard, without it being displayed on the screen; this is achieved by deleting the instruction, RST 16, from the routine.

The method described above is the easiest way of carrying out the often required process of inputting a character and displaying it on the screen. The Sinclair ROM includes a number of other routines which can be used for inputting from the keyboard. If you are interested in these routines, I will leave it to you to experiment with the routines in the ROM.

6.2 Character codes

I must stress that the input routine given in the last section loads into the accumulator the ASCII code for the character whose key is pressed. Appendix E shows the normal character codes which can be input from the keyboard using this input routine.

It must be realized, when inputting numbers, that the character code for the number is not the same as the value of the number. If numbers are input to the computer, used for arithmetic, and then output from the computer, we must make sure that we use the character code and the value in the correct places. All input and output processes use the character code and any arithmetic uses the value of the character. When using ASCII codes, as used by the Spectrum, the character codes of the digits, 0 to 9, can be converted to their numeric value by subtracting 48 from the character code. Remember to add on 48 again when outputting the results.

6.3 Number input

So far we have only considerd how single-digit numbers can be input or output. This is obviously very restricted and this section will look at routines for inputting and outputting numbers with several digits. Inputting a number with more than one digit is not as straightforward as it would seem, even when the input is restricted to whole numbers.

If the number to be input is a three-digit number, such as 164, this will, of course, involve pressing the three keys '1', '6', and '4'. this could be carried out by calling the input and output subroutine three times and loading the character codes as they are input into successive memory locations. When all of the digits of the number have been input, this is usually indicated by the input of another character such as a comma or ENTER; the separate character codes must be converted into a value in a single register or memory location.

The first stage in this conversion is to change the character codes into the numeric value for each digit. The calculation which must now be carried out is to multiply the first digit by 100, multiply the second digit by 10, and then add these products to the value of the third digit. For the number 164 the calculation is:

$1\times100+6\times10+4$

A more efficient method is to carry out the calculation:

$(1\times10+6)\times10+4$

After the character codes have been converted to their digit values, the fiirst digit is multiplied by 10, the second digit is added to this product, the result is multiplied by 10, and the third digit is added to the product. The big advantage of this method is that it can be extended to cover the input of a number with any number of digits.

We have already looked at some simple arithmetic in an earlier chapter and you will recall that we could only carry out addition and subtraction. There are no direct instructions in the assembly language of the Spectrum for multiplication or division. A simple means of carrying out multiplication is repeated addition: for example, 5×3 is the same as 5+5+5 and 6×10 is 6+6+6+6+6+6+6+6+6+6. This appears to be a very crude method but, using a loop, it is simple to program and remarkably quick.

Figure 6.2 is a subroutine which will input a number and convert it into a value stored in memory. Although the routine is general and will input a number with as many digits as required, the final value is stored in a single memory location which means that the maximum value which can be input is 255. The routine accepts any input as a digit until the ENTER key is pressed; it could easily be extended to check that the key pressed is a digit or ENTER.

```
 10 REM go
 20 REM org 23760
 30 REM !set up storage
 40 REM Number;defb 0
 50 REM !start of number input
 60 REM Numin;call Cinout
 70 REM !check for ENTER
 80 REM cp 13
 90 REM ret z
100 REM !change code to value
```

```
 110 REM sub 48
 120 REM ld b,a
 130 REM !recall previous total
 140 REM ld a,(Number)
 150 REM !multiply by ten
 160 REM call Multen
 170 REM add a,b
 180 REM !save new number
 190 REM ld (Number),a
 200 REM jr Numin
 210 REM !
 220 REM !subroutine to multiply
 by ten
 225 REM !
 230 REM Multen;add a,a;! times
2
 240 REM ld c,a
 250 REM add a,a;! times 4
 260 REM add a,a;! times 8
 270 REM add a,c;!times 8 + time
s2
 280 REM ret
 285 REM !
 290 REM !input subroutine
 295 REM !
 300 REM Cinout;call 4274
 320 REM cp 208
 330 REM jr z,Cinout
 335 REM push af
 340 REM rst 16
 380 REM pop af
 390 REM ret
 400 REM finish
```

Figure 6.2

Outputting the value in a register is essentially the reverse of the above process but it is even more complex than input. Figure 6.3 is a subroutine to output the value in the accumulator.

```
  10 REM go
  15 REM org 23760
  20 REM !subroutine to print
number in A register
  25 REM !save value in A
  30 REM ld b,a
  35 REM !print to top of screen
  40 REM ld a,2
  50 REM call 5633
  55 REM !start of main routine
  60 REM ld a,b
  70 REM ld b,0
  80 REM Loopa;sub 100
  90 REM jp m,Hundreds
 100 REM inc b;! count no. of 10
0's
 110 REM jr Loopa
 115 REM !print no. of 100's
 120 REM Hundreds;add a,100
 130 REM ld c,a;!save rest of nu
mber
 140 REM ld a,b
 150 REM cp 0;! no 100's
 160 REM jr z,Continue
 170 REM add a,48;!change to cod
e
 180 REM rst 16
 190 REM ld d,1;! set flag to sh
ow hundreds digit printed
 200 REM Continue;ld b,0
 210 REM ld a,c
 220 REM Loopb;sub 10
 230 REM jp m,Tens
 240 REM inc b;!count no. of 10'
s
 250 REM jr Loopb
 255 REM !print no. of 10's
 260 REM Tens;add a,10
 270 REM ld c,a;!save rest of nu
mber
 280 REM ld a,b
 290 REM cp 0;!any tens
 300 REM jr nz,Next
 310 REM ld a,d
 320 REM cp 1;!need to print 0?
 330 REM jp nz,Digits
 340 REM ld a,b
 350 REM Next;add a,48;! code
 360 REM rst 16
 370 REM Digits;ld a,c
 380 REM add a,48
 390 REM rst 16
 400 REM ret
 410 REM finish
```

Figure 6.3

6.4 Negative numbers

All the numbers we have used till now have been positive integers (whole numbers) and the number input routine given in the last section only takes positive numbers into account. The computer can, of course, be programmed so that it will recognize positive and negative numbers. The simplest method of getting a negative number is to use the instruction:

```
NEG
```

which will take the two's complement of the value in the accumulator. If the value in the accumulator is a number between 0 and 127, this instruction will turn the value into the signed integer representation of a number between 0 and −127, as shown in Chapter 2. For example, if the accumulator contains the binary equivalent of the number 45 and the instruction NEG is executed the accumulator will contain the signed integer representation of the number −45. Figure 6.4 shows the contents of the accumulator in binary before and after the NEG instruction.

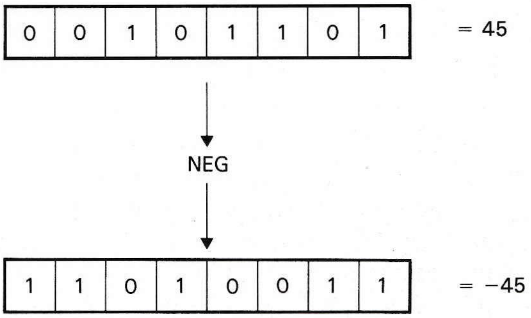


Figure 6.4

When using signed integer representation the value in a single eight-bit register or memory location must be in the range −128 to +127. This restriction on the range of values also applies to the results of any arithmetic operations carried out on signed numbers.

It is now necessary to modify our number input routine so that it will input negative or positive numbers. The main modification is due to the first character which will be either a '−' character or a '+' character or a digit. If the first character is a '+' or a digit then the number will be input as shown in the previous section, but if the first character is a '−' then after the digits have been input and converted into a positive value a NEG instruction will be executed to convert it to a negative value. I will leave the actual modification of the routine as an exercise for you to carry out.

6.5 Carry and overflow

When carrying out arithmetic using eight-bit data all numbers must be in the range 0 to 255, if we are using unsigned integer representation, or −128 to +127, if they are signed integers. We shall now look at how we can check whether our results are staying within the allowable ranges.

Looking first at the addition of unsigned integers, Figure 6.5 shows two examples of the addition of eight-bit numbers. In the first example the correct result can be expressed as an eight-bit number and therefore our arithmetic can be carried out correctly by the computer. The second example again shows the addition of two eight-bit numbers, but the correct result of the addition needs nine bits and so it cannot be contained in an eight-bit register and the computer produces the wrong result. So that we can test whether the correct result has been produced, an addition which produces a '1' in the ninth-bit position puts this bit into one of the bit positions in the flag, or F, register. This bit is the carry flag and it is set to one if an addition, or subtraction, produces a ninth bit; otherwise it is set to zero. Figure 6.6 gives examples of a ninth bit, or carry, produced by subtraction. The value in this bit can be tested by a JP instruction testing for:

```
                 00101101       45
               + 10011001     + 153
                 --------      ---
                 11000110      198  Right

                 10010001      145
               + 10011001     + 153
                 --------      ---
     Carry = 1   00101010       42  Wrong
```

Figure 6.5

```
                 10011001      153
               - 10010001    - 145
                 --------      ---
                 00001000        8  Right

                 10010001      145
               - 10011001    - 153
                 --------      ---
     Carry = 1   11111000      248  Wrong
```

Figure 6.6

C carry flag set to 1

or

NC carry flag set to 0

The following short program segment shows a typical use:

```
ADD A,B
JP C,ERROR
ERROR; ! Start of an error routine
```

Carry (C) and no carry (NC) can also be used with relative jump (JR) instructions.

As well as addition and subtraction instructions, the carry flag is also affected by shift and rotate instructions; these will be discussed in Chaper 10.

Because the carry flag is so important to the computer's arithmetic, especially when using numbers held in more than one byte, there are two instructions which allow the value of the carry flag to be changed directly. They are SCF, which sets the carry flag to one, and CCF, which sets the carry flag to the opposite of its present value.

The flag register also contains a bit known as the overflow flag, which is used to indicate that the result of an arithmetic operation is outside the range allowed for signed numbers. When carrying out addition overflow can only occur if both values are positive or both values are negative and subtraction can only produce overflow if one number is positive and the other negative. Figure 6.7 shows examples of addition and subtraction which produce overflow.

```
                01101010       + 106
              + 01011110       +  94
                --------       -----
                11001000       -  56   Wrong answer (overflow)

                00011010       +  26
              + 01011110       +  94
                --------       -----
                01111000       + 120   Right answer (no overflow)

                10010110       - 106
              + 10100010     + -  94
                --------       -----
Carry = 1       00111000       +  56   Wrong answer (overflow)

                01101010       + 106
              - 10100010     - -  94
                --------       -----
Carry = 1       11001000       -  56   Wrong answer (overflow)
```

Figure 6.7

The overflow flag is set to one if overflow occurs; otherwise it is set to zero. The value in the overflow flag is tested by a conditional jump instruction. The overflow flag also has a second function, indicating the parity of a byte, and the mnemonic for the condition refers to this use. Do not worry about parity; we are only interested in the indication of overflow.

The instructions which test the overflow flag are:

JP PE, label

JP PO, label

where PE means test for no overflow and PO means test for overflow. There are no relative jump (JR) instructions which test the overflow flag.

Figure 6.8 is a subroutine which reads two values from memory

```
  10 REM go
  20 REM org 23760
  25 REM !set up storage
  30 REM Num1;defb 0
  35 REM Num2;defb 0
  40 REM Sign;defb 0
  45 REM Result;defb 0
  50 REM !start of program
  60 REM ld hl,Num1
  70 REM ld b,(hl);! 1st number
  80 REM inc hl;!point to 2nd nu
mber
  90 REM ld c,(hl);! 2nd number
 100 REM ld a,(Sign)
 110 REM cp 43;! test +
 120 REM jr z,Addit
 130 REM cp 45;! test -
 140 REM jr z,Subit
 150 REM jr Error
 155 REM !add routine
 160 REM Addit;ld a,b
 170 REM add a,c
 180 REM ld (Result),a
 185 REM !test for overflow
 190 REM jp po,Error
 200 REM ret
 205 REM !
 210 REM !subtract routine
 220 REM Subit;ld a,b
 230 REM sub c
 240 REM ld (Result),a
 245 REM !test for overflow
 250 REM jp po,Error
 260 REM ret
```

```
265 REM !
270 REM !error routine
280 REM Error;ld a,2
290 REM call 5633
300 REM ld a,69;! E
310 REM rst 16
320 REM ld a,114;! r
330 REM rst 16
332 REM ld a,114;! r
335 REM rst 16
340 REM ld a,111;! o
350 REM rst 16
360 REM ld a,114;! r
370 REM rst 16
380 REM ret
390 REM finish
```

Figure 6.8

and adds, or subtracts, them according to the character code held in a third memory location. The result is stored in another memory location. If overflow occurs, the word ERROR is output. Figure 6.9 shows a simple BASIC program which can be used to test this subroutine.

```
   1 REM 00000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
000
  10 INPUT "First number?",a
  20 INPUT "Second number? ",b
  30 INPUT "A(dd) or S(ubtract)?
 ",x$
  40 POKE 23760,a
  50 POKE 23761,b
  60 IF x$(1)="A" THEN  POKE 237
62,43: GO TO 90
  70 IF x$(1)="S" THEN  POKE 237
62,45: GO TO 90
  80 POKE 23762,0
  90 RANDOMIZE USR 23764
 100 PRINT PEEK 23763
```

Figure 6.9

6.6 The EQU instruction

The EQU instruction is another pseudo-operation like the DEFB instruction discussed in the last chapter. Although it does not produce any machine code instructions when the program is translated it is very useful to the assembly language programmer.

The form of the instruction for the ZX Spectrum Machine Code Assembler is

```
EQU address name
```

and its effect is to allow the programmer to give a label to memory locations even if outside the machine code program. This produces programs which are easier to read and understand and is particularly important when writing fairly long programs in several sections. The following short program segment shows a typical use.

```
EQU 5633 Channel
____
____
____
____
LD A, 2
CALL Channel
LD A,13
RST 16
```

In this segment the location whose address is 5633 is given the name 'Channel' and it can then be referred to by name rather than by using the address 5633. The use of the name makes the purpose of the instructions much clearer that it would be using the address.

6.7 Program

Write a program, using the subroutines developed in this chapter, which requests two numbers to be input by first outputting a prompt such as:

INPUT FIRST NUMBER
then INPUT SECOND NUMBER

The second number should be a positive number, but the first can be positive or negative. Multiply the two numbers together using repeated addition. If overflow occurs the word ERROR should be output; otherwise the result should be output.

7 SIXTEEN-BIT NUMBERS

7.1 Register pairs

Most of the data in the Spectrum computer is moved around and used as eight-bit numbers. There are times, however, when it is much more convenient to have data processed as sixteen-bit numbers. In particular, when the processing is concerned with the determination of an address for a memory location, 16-bit processing is almost essential. Apart from the specific 16-bit registers, which are used for special purposes and will be discussed later, all of the general purpose, eight-bit registers can be used in pairs, to form the 16-bit registers BC, DE, and HL. The HL register pair is often used as a 16-bit accumulator and is only rarely used as two, separate eight-bit registers.

All of the 16-bit registers can be loaded with a numeric value directly using the instruction:

```
LD dd,nn
```

where nn is a number in the range 0 to 65 535 and dd is one of the 16-bit registers BC, DE, HL, IX, IY, or SP.

The only instructions which allow data to be copied from one 16-bit register to another are instructions involving the SP register. This register will be discussed in detail in Chapter 8. Data can be copied from one register pair to another by treating them as single eight-bit registers. For example, to copy the value in the HL register to the BC register the instructions:

```
LD B,H
LD C,L
```

would be used. The values in the HL and DE register pairs only, can be exchanged using the instruction:

```
EX DE,HL
```

7.2 Sixteen-bit data in memory

There are instructions which allow data to be moved between the memory and the 16-bit register pairs. However, since each location in memory can only hold an eight-bit value, 16-bit values have to be held in two successive locations. When a 16-bit value is moved from a register pair to memory, the last eight bits are moved to the first of the locations and the first eight bits are moved to the second of the locations. Figure 7.1 illustrates this process. Moving from memory to a register pair is carried out in a similar fashion. The instructions for carrying out these moves are:

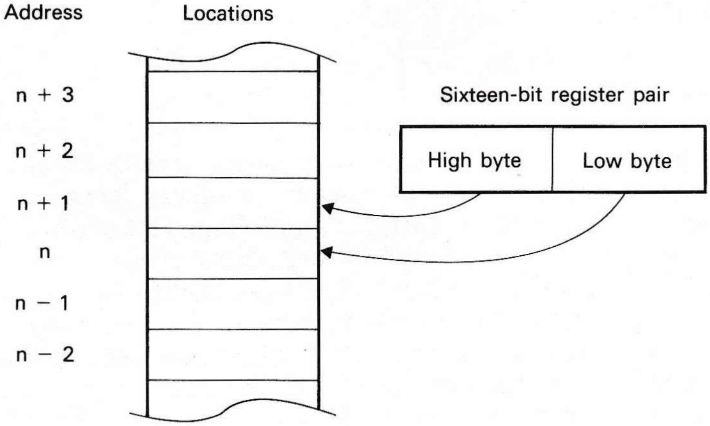


Figure 7.1

LD dd,(nn)   and   LD (nn),dd

where dd is one of the register pairs and nn is the address of the first of the two memory locations.

7.3 More from the keyboard

In the last chapter we saw how to use the subroutine in the Spectrum ROM to input a character code into the accumulator. All of the programs you would ever want to write in assembly language can be written using just this subroutine to decode any input from the keyboard. At times you may wish to use the keyboard rather differently from the normal way of allocating a separate code to each key. For example, in a games program you might decide that any key on the left of the keyboard will move left and any on the right will move right. In order to obtain this flexibility we need to examine more closely how the microprocessor obtains data from the keyboard. All information which comes into the central processor, with the exception of data from memory, is input using one of the special input instructions. Input from the keyboard will use the instruction:

```
IN A,(n)
```

When this instruction is encountered the value n, known as the port or device number, gives the information which tells the system which external device, or even which part of a device, is to be examined for data. The data, if any, are then put into the accumulator. This instruction only inputs one byte of data at a time.

Now let us look at how the keyboard of the Spectrum looks to the microprocessor and how we can write programs to control it. When a key is pressed two signals are produced. One signal is sent to port 254 and can be input to the microprocessor by using the instruction IN A,(254). This signal is produced by treating the keyboard as eight half-rows each consisting of five keys. Pressing any of the keys in a row of five sets one of the bits in the eight-bit signal to zero. No key being pressed sets all of the bits to one. Figure 7.2 shows the half-rows and the bit number affected by each group of keys. You can see that a key on the left of the keyboard will set one of the bits 0, 1, 2, or 3 to zero and a key on the right will set one of bits 4 to 7 to zero.

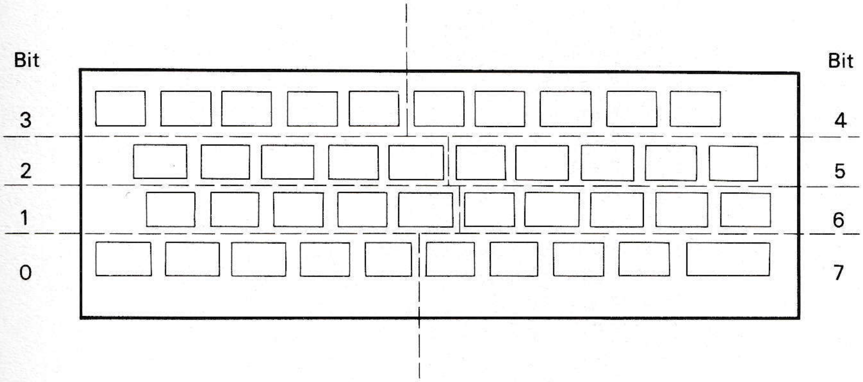


Figure 7.2

Figure 7.3 is a short program which prints a character to the centre of the screen and moves it left or right depending upon which half of the keyboard is pressed.

```
   1 REM 00000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
000
  10 REM go
  15 REM org 23760
  18 REM !scan left or right of
 keyboard
  20 REM Keyscan;in a,(254)
  25 REM cpl
  30 REM ld b,a;! temp store
  35 REM and 15;!left side?
  40 REM jr nz,Left
  45 REM ld a,b
  50 REM and 240;!right side?
  55 REM jr nz,Right
  60 REM jr Keyscan
  65 REM !
  70 REM Left;ld a,2
  75 REM call 5633
  80 REM ld a,76;! L
  85 REM rst 16
  90 REM ld a,69;! E
  95 REM rst 16
 100 REM ld a,70;! F
 105 REM rst 16
 110 REM ld a,84;! T
 115 REM rst 16
 117 REM ld a,13;! ENTER
 118 REM rst 16
 120 REM ret
 125 REM !
 130 REM Right;ld a,2
 135 REM call 5633
 140 REM ld a,82;! R
 145 REM rst 16
 150 REM ld a,73;! I
 155 REM rst 16
 160 REM ld a,71;! G
 165 REM rst 16
 170 REM ld a,72;! H
 175 REM rst 16
 180 REM ld a,84;! T
 185 REM rst 16
 190 REM ld a,13;! ENTER
 195 REM rst 16
 200 REM ret
 210 REM finish
```

Figure 7.3

7.4 Computer music

One of the features of the Spectrum is its ability to produce output through its internal loudspeaker. The sound is produced by sending electrical pulses to the loudspeaker; this can be done directly from machine code by sending data to the output port which is connected to the loudspeaker, or more simply by using the sound subroutine in the Spectrum ROM.

The use of the output port will be discussed in a later chapter. This present section will look at the use of the ROM subroutine. The subroutine starts at address 949 in memory and is brought into operation by the instruction:

CALL 949

Before the subroutine can be used values must be put into the HL and DE register pairs. The value in the HL register controls the time interval between the pulses sent to the loudspeaker, in this way controlling the pitch of the note which is produced. The lower the value in HL, the higher the note produced because the interval between pulses is shorter. The value in the DE register controls the length of the note—the higher the value in DE, the longer the note produced.

Figure 7.4 shows a very simple program which will produce a single sound. Experiment with different values in DE and HL; the results can be surprising. You should also experiment with a program to change the values in DE and HL using loops; some very interesting and unusual sounds can be produced.

```
 1 REM 000000000000
10 REM go
20 REM org 23760
30 REM ld hl,650;! pitch
40 REM ld de,100;! length
50 REM call 949;!sound routine
60 REM ret
70 REM finish
```

Figure 7.4

7.5 Addressing modes

When we are writing programs in BASIC we write program statements such as:

LET A=B+5

without thinking whether we are referring to the value in location A or to the address of location A. However, when writing assembly language programs we must be sure that we specify the correct data in our instructions. The methods used for specifying data in instructions are known as addressing modes. In this section I would like to recap on the addressing modes we have met so far, and clarify the ways in which they specify data.

The Z80 microprocessor has 10 different addressing modes; we have already used several of these modes.

The immediate addressing mode has the actual value of the data in the instruction.

The register addressing mode uses the name of a register as the operand of the instruction. The data have the value in that register.

The extended addressing mode uses data from memory. The operand of the instruction is the address of the memory location which contains the data. In assembly language, extended addressing is shown by brackets around the operand. This mode is also called indirect addressing.

The relative addressing mode is only used for relative jump instructions, with or without conditions. The operand for these instructions is the displacement or the number of memory locations to the instruction to be executed if the jump is made.

The implied addressing mode does not use anything other than the instruction itself to indicate the data to be used. Instructions using this addressing mode do not have a separate operand.

Figure 7.5 gives examples of these addressing modes and shows how they are stored in memory.

Instruction	Machine code	Addressing
SUB 45	214, 45	Immediate
INC B	4	Register
LD A,(32000)	58, 00, 125	Extended
JR C,LOOP	56, 54	Relative

Figure 7.5

There are two further addressing modes to be discussed. The register indirect addressing mode uses data from memory, like the extended addressing mode, but the operand in the instruction is a register pair which contains the address of the memory location holding the data. For example, the instruction:

LD A,(HL)

takes the value in the HL register pair as the address of a memory location and then puts a copy of the value in that memory location. This is shown in Fig. 7.6. When loading data into the accumulator any of the 16-bit register pairs may be used to hold the address of a memory location, but for loading any of the other single eight-bit registers only the HL register pair can be used to

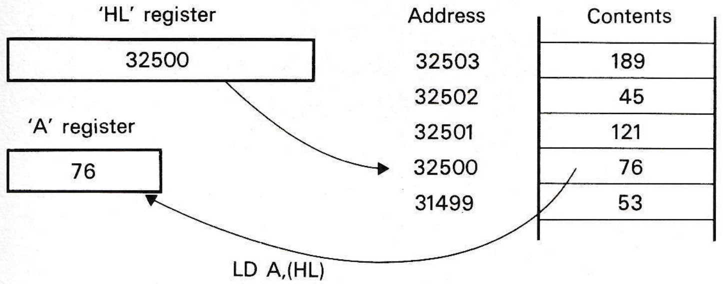


Figure 7.6

hold the memory address. This addressing mode can also be used for arithmetic and jump instructions such as:

ADD A,(HL)
SUB (HL)
JP (HL)

When used for arithmetic or jump instructions only the HL register pair can be used to hold the address of the memory location. Figure 7.7 shows some examples of this addressing mode. As well as moving data from memory to the accumulator or any other eight-bit register, this addressing mode can also be used to move data from registers to memory. An example of this is shown in the figure. The extra flexibility gained from using the HL register pair is the reason that it is used as the main memory pointer.

The immediate extended addressing mode is an extension of the immediate addressing mode. Whereas immediate addressing uses eight-bit values, immediate extended addressing uses 16-bit values. This mode is used to put a value directly into one of the 16-bit register pairs. For example, the instruction:

LD BC,15200

will put the value 15 200 into the BC register pair. Any of the 16-bit registers BC, DE, HL, SP, IX, and IY may be loaded with a value. Figure 7.8 is a short program segment which shows the use of immediate extended and register indirect addressing modes.

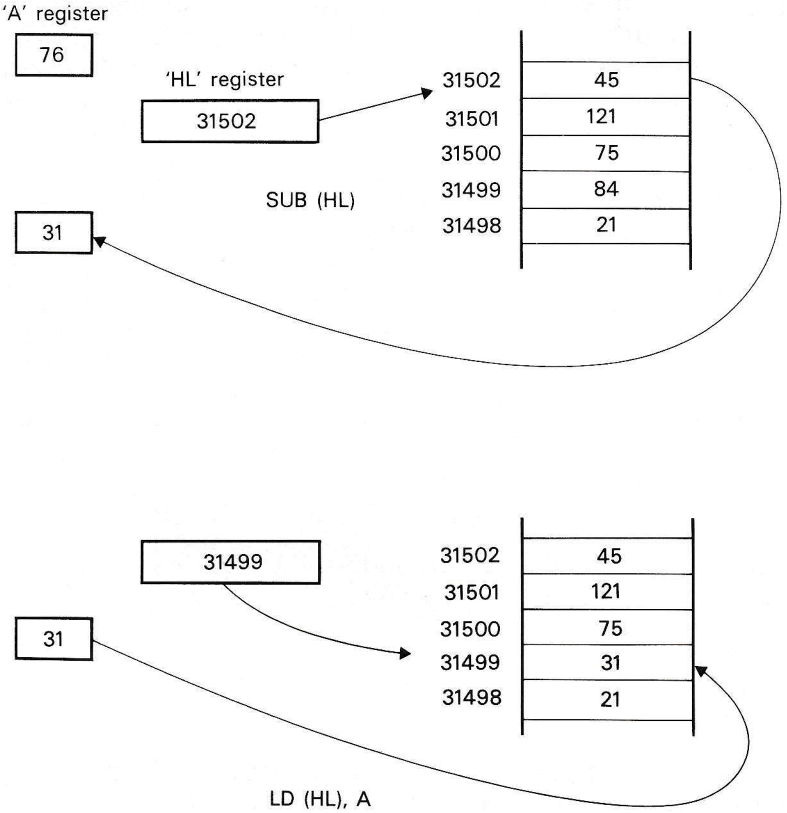

'A' register
76
'HL' register
31502
31502
45
31501
121
31500
75
31499
84
31498
21
SUB (HL)
31
31499
31502
45
31501
121
31500
75
31499
31
31498
21
31
LD (HL), A

Figure 7.7

```
  10 REM go
  20 REM org 23760
  30 REM !addressing modes
  40 REM ld hl,31500;!immediate
extended
  50 REM ld a,(hl);!register ind
irect
  60 REM ld hl,31520
  70 REM sub (hl);!register indi
rect
  80 REM ld (hl),a;!register ind
irect
  90 REM ld hl,31522
 100 REM jp (hl);!register indir
ect
 110 REM ret
 120 REM finish
```

Figure 7.8

7.6 Program

Write a program which will change the number keys into a simple musical keyboard. Pressing different keys will play different notes. You will need to experiment to give a range of notes which sound good. Pressing and holding a key should give a sustained note; this can be achieved by making the duration of each note very short and looping back to the keyboard input routine. You will need to modify the input routine to take out the echo.

8 DOING IT AGAIN

8.1 Loops

Any program which repeats a section of program more than once is using a loop. Probably the most important characteristic of any computer is its ability to do the same thing many times without making mistakes—in other words, its ability to carry out a program loop. We can recognize different types of loop by the method used to determine when to stop executing the loop.

The simplest type of loop is the endless, or infinite loop. Figure 8.1 shows a simple endless loop, which produces a screen display. Endless loops should not normally be used because they can only be stopped by switching off the computer.

```
 10 REM go
 20 REM org 23760
 30 REM equ 5633 Chanopen
 40 REM ld a,2
 50 REM call Chanopen
 60 REM ld a,31
 70 REM Loop;inc a
 75 REM ld b,a
 80 REM rst 16
 85 REM ld a,b
 90 REM jp Loop
100 REM ret
110 REM finish
```

Figure 8.1

A useful method of stopping a loop is to use a test for a specific condition as the means of ending the loop. The program in Figure 8.2 inputs characters from the keyboard and puts them into successive memory locations until the ENTER key is pressed. The instructions in the loop are repeated until the input routine inputs the character code 13, the code for ENTER. The compare instruction which follows the input routine will produce a zero result when ENTER is pressed and the conditional jump, JP NZ, Input, will be ignored and the program will continue with the instruction labelled 'Continue'.

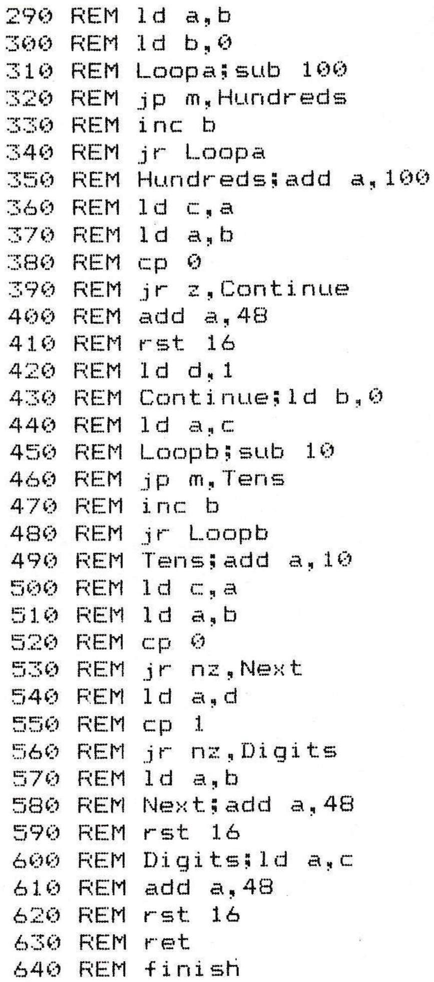
```
 10 REM go
 20 REM org 23760
 30 REM ld hl,32499
 40 REM !store data from 32500
 50 REM Input;inc hl
 60 REM call Cinout
 70 REM ld (hl),a
 80 REM cp 13;!test for ENTER
 90 REM jr nz,Input
100 REM Continue;ret
110 REM !
120 REM Cinout;call 4264
140 REM cp 208
150 REM jr z,Cinout
160 REM push af
170 REM rst 16
180 REM pop af
190 REM ret
240 REM finish
```

Figure 8.2

8.2 Counting loops

One of the easiest ways of ending a loop is to execute it a set number of times. This is called a counting loop and Fig. 8.3 is a program which shows a simple example of a counting loop. This program uses the B register as a counter for the number of times

```
 10 REM go
 20 REM org 23760
 30 REM ld b,5;!do it 5 times
 40 REM ld hl,32500
 50 REM Loop;call Cinout
 60 REM sub 48;!code to value
 70 REM add a,(hl)
 80 REM ld (hl),a;!save result
 90 REM djnz Loop
100 REM call Numout
110 REM ret
120 REM !
130 REM Cinout;call 4264
150 REM cp 208
160 REM jr z,Cinout
170 REM push af
180 REM rst 16
230 REM pop af
240 REM ret
250 REM !
260 REM Numout;ld b,a
270 REM ld a,2
280 REM call 5633
```



```
290 REM ld a,b
300 REM ld b,0
310 REM Loopa;sub 100
320 REM jp m,Hundreds
330 REM inc b
340 REM jr Loopa
350 REM Hundreds;add a,100
360 REM ld c,a
370 REM ld a,b
380 REM cp 0
390 REM jr z,Continue
400 REM add a,48
410 REM rst 16
420 REM ld d,1
430 REM Continue;ld b,0
440 REM ld a,c
450 REM Loopb;sub 10
460 REM jp m,Tens
470 REM inc b
480 REM jr Loopb
490 REM Tens;add a,10
500 REM ld c,a
510 REM ld a,b
520 REM cp 0
530 REM jr nz,Next
540 REM ld a,d
550 REM cp 1
560 REM jr nz,Digits
570 REM ld a,b
580 REM Next;add a,48
590 REM rst 16
600 REM Digits;ld a,c
610 REM add a,48
620 REM rst 16
630 REM ret
640 REM finish
```

Figure 8.3

the loop is executed. The instruction DJNZ decreases the value in the B register by one and causes a jump as long as this value is not zero. When the value in B becomes zero the loop has been executed the required number of times and the program will continue with the next instruction.

The DJNZ instruction stands for 'decrement and jump if non zero' and it is equivalent to the two separate instructions:

DEC B

JR NZ, label

The DJNZ instruction can only be used with the B register. Although other registers, or even values in memory, can be used as counters for loops, it is usual to use the B register because of the DJNZ instruction.

The program in Fig. 8.3 is written with a set value in the B register. It is better, when possible, to make your routines as general as possible, so that they can be used with other programs. The program would have been better if the loop counter value had been taken from a memory byte. The value can be put in memory by a main program which uses the routine.

Sometimes the loop value counter will also be used as a data value in your program. Figure 8.4 shows a short program which calculates the sum of successive numbers. Note that in this

```
 10 REM go
 30 REM call Cinout
 40 REM sub 48
 50 REM ld b,a
 60 REM ld a,0
 70 REM Loop;add a,b
 80 REM djnz Loop
 90 REM call Numout
100 REM ret
120 REM !
130 REM Cinout;call 4264
150 REM cp 208
160 REM jr z,Cinout
170 REM ld b,a
180 REM rst 16
230 REM ld a,b
240 REM ret
250 REM !
260 REM Numout;ld b,a
270 REM ld a,2
280 REM call 5633
290 REM ld a,b
300 REM ld b,0
310 REM Loopa;sub 100
320 REM jp m,Hundreds
330 REM inc b
340 REM jr Loopa
350 REM Hundreds;add a,100
360 REM ld c,a
370 REM ld a,b
380 REM cp 0
390 REM jr z,Continue
400 REM add a,48
410 REM rst 16
420 REM ld d,1
430 REM Continue;ld b,0
```

```
440 REM ld a,c
450 REM Loopb;sub 10
460 REM jp m,Tens
470 REM inc b
480 REM jr Loopb
490 REM Tens;add a,10
500 REM ld c,a
510 REM ld a,b
520 REM cp 0
530 REM jr nz,Next
540 REM ld a,d
550 REM cp 1
560 REM jr nz,Digits
570 REM ld a,b
580 REM Next;add a,48
590 REM rst 16
600 REM Digits;ld a,c
610 REM add a,48
620 REM rst 16
630 REM ret
640 REM finish
```

Figure 8.4

program the accumulator is set to zero before being used to hold the sum. This is because the value, in registers or memory locations, will not necessarily be zero unless it is set to zero by the programmer.

When the DJNZ instruction is used, the loop is executed until the loop counter reaches the value zero. It is often more convenient to use a loop counter until it reaches some value other than zero. When this happens, a compare instruction is used to detect the end of the loop. The compare instruction can only be used with the accumulator so the accumulator is used as the loop counter. Figure 8.5 is a short program to show this use of the accumulator.

8.3 What is a stack?

A stack is a particular method of storing data in successive memory locations. The most important feature of a stack is that data can only be recovered from it in the reverse order to that in which it was stored. In other words the first item which can be recovered is the last item to be stored. Another term for a stack is a last-in-first-out list.

Because a stack is a very useful method of storing data there is a special register in the microprocessor to control stacks and

```
  10 REM go
  20 REM org 23760
  30 REM !loop using accumulator
 as a counter
  40 REM ld hl,32500;!memory poi
nter
  50 REM ld a,10;!start value
  60 REM Loop;ld b,(hl)
  70 REM inc hl
  80 REM ld (hl),b
  90 REM add a,b
 100 REM cp 104
 110 REM jr nz,Loop
 120 REM ret
 130 REM finish
```

Figure 8.5

instructions for storing and recovering data from stacks. Stacks are used extensively in the ROM by the computer's operating program.

Figure 8.6 shows how stacks are stored in the memory of the Spectrum. The diagram shows that the stack is stored upside down, the top of the stack being at a lower address than the bottom. It is stored like this so that with the stack starting at the

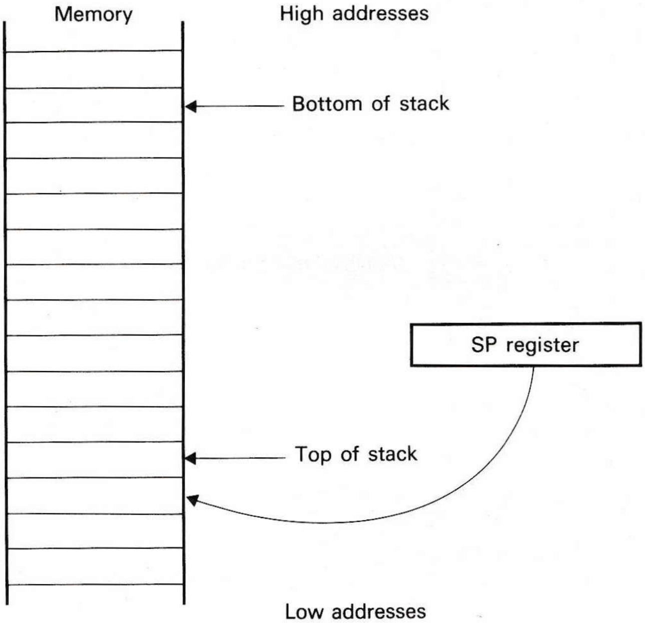

Memory
High addresses
Bottom of stack
SP register
Top of stack
Low addresses

Figure 8.6

top of memory the size of the stack can continue increasing until the memory is full.

Each data item in the stack is two bytes long. When data are added to or removed from the stack, the value in the stack pointer, which always points to the first empty address in the stack, is decreased or increased by two.

Stacks are used for the temporary storage of data.

8.4 Uses of stacks

One of the main uses of the stack by the computer's operating program is to control the return from a subroutine to the main program. You will need to refer to Fig. 8.7 in conjunction with the

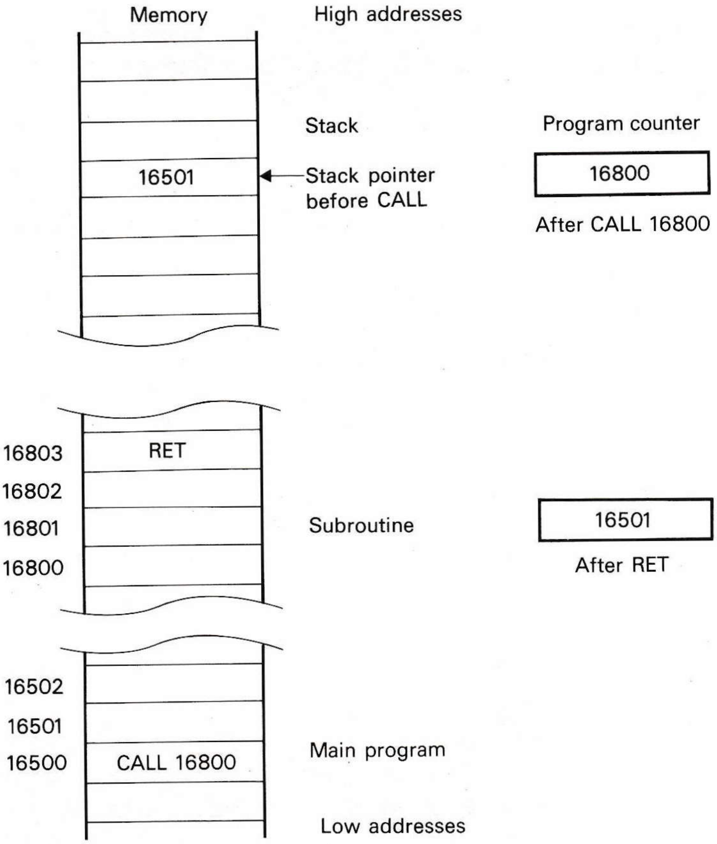

Memory
High addresses
Stack
Program counter
16501
Stack pointer before CALL
16800
After CALL 16800
16803
RET
16802
16801
Subroutine
16501
16800
After RET
16502
16501
16500
CALL 16800
Main program
Low addresses

Figure 8.7

following description. When a program reaches a CALL instruction the value in the program counter, which is the address of the next instruction, is put on the stack and the program counter is then loaded with the address of the first instruction of the subroutine. This has the effect of starting the subroutine. At the end of the subroutine, the return instruction takes the top value off the stack and puts it into the program counter. The figure shows a subroutine at location 16 800 which jumps back to the main program at the instruction at location 16 501. Before the subroutine is called the stack pointer points to the first empty location. When the subroutine reaches the return instruction, the address of the correct instruction must now be at the top of the stack; if the stack has been used in the subroutine then the programmer must ensure that the same number of data items are added and removed from it by the subroutine.

However, it is more important to see how you can use stacks in your own programs. A common use of the stack is to save the values held in the microprocessor's registers while a subroutine is being used. Unless a register is used to transfer information between the subroutine and the main program, all subroutines should save in memory the values in any registers which will be used by the subroutine. At the end of the subroutine, just before returning to the main program, the stored values can be retrieved and replaced in the correct registers. Using this technique, subroutines can be used by any program without having to worry about register values changing during the running of the subroutine. When using the stack in this way it is important that data are taken off the stack in the reverse order from when they were stored.

There are several instructions which allow the programmer to use stacks. On the Spectrum, data are added to the stack two bytes at a time; this is known as pushing data onto the stack, and the instruction is:

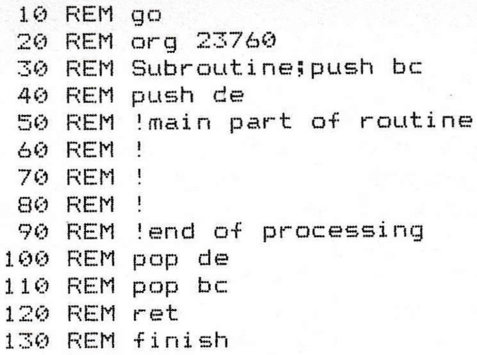
```
PUSH rp
```

where rp is any of the 16-bit register pairs AF, BC, DE, HL, IX, or IY. Recovering data from the stack is called popping the stack, and the instruction is

```
POP rp
```

where rp is any 16-bit register pair. The instructions PUSH and POP not only store data in the area of memory occupied by the stack but also change the values in the SP register.

Figure 8.8 shows how the stack would be used by a subroutine



```
 10 REM go
 20 REM org 23760
 30 REM Subroutine;push bc
 40 REM push de
 50 REM !main part of routine
 60 REM !
 70 REM !
 80 REM !
 90 REM !end of processing
100 REM pop de
110 REM pop bc
120 REM ret
130 REM finish
```

Figure 8.8

which uses the B, C, and D registers to store the values in these registers during the execution of the subroutine.

For most normal requirements, the programmer will use the stack set up by the computer's operating system for storing data. But a stack is a useful method of storing data and a programmer will sometimes wish to set up a separate stack for data storage. This is done by putting the address at the top of an area of unused memory into the SP register. For example, the instruction:

```
LD SP,20000
```

will start a stack from location 20 000 in memory.

Because data cannot be moved directly from one 16-bit register pair to another 16-bit register pair, it is often easiest to move data using the stack as an intermediate store.

There are also three instructions which allow data stored in a 16-bit register to be exchanged directly with the last item on the stack. The form of the instruction is:

```
EX (SP),rp
```

where rp is one of the HL, IX, or IY 16-bit registers.

8.5 Displaying messages

Most programs at some stage in their processing need messages sent to the screen or printer. In BASIC this is done using PRINT or LPRINT statements with the message enclosed in quotation marks. However, in assembly language the display of a message needs to be treated in two steps. The first step consists of storing the message in the program and then, separately, a section of program sends the message to the display, or printer.

The first step is accomplished by using one of those special instructions called pseudo-operations; this time the instruction is:

DEFS

The instruction is usually preceded by a label and is followed by the text of the message. A typical instruction is:

TITLE;DEFS This is a program

The effect of this instruction is to store the message as a string of character codes in successive memory locations, starting at the location labelled TITLE. To output the message to the display the print subroutine, called by the instruction RST 16, is used inside a loop which counts how many characters are to be displayed. Figure 8.9 is a short program which displays the text given in the example DEFS instruction. Although this routine works perfectly well it would be better if it was made more general so that it could be used to print out any text. Figure 8.10 is a modified version of the previous routine. It takes the starting address of the text from the HL register, which is loaded with this

```
  10 REM go
  20 REM org 23760
  30 REM Title;defs This is a pr
ogram
  40 REM push af
  45 REM push bc
  50 REM push hl
  55 REM ld a,2
  60 REM call 5633;!open channel
  65 REM !
  70 REM ld b,17;!number of lett
ers
  75 REM ld hl,Title
  80 REM Loop;ld a,(hl);!put let
ter in A register
  85 REM rst 16;!print it
  90 REM inc hl;!point to next l
etter
  95 REM djnz Loop
 100 REM pop hl
 105 REM pop bc
 110 REM pop af
 120 REM ret
 130 REM finish
```

Figure 8.9

```
   10 REM go
   20 REM org 23760
   30 REM Title;defs This is a pr
ogram
   40 REM defb 0
   45 REM push af
   50 REM push hl
   55 REM ld a,2
   60 REM call 5633;!open channel
   65 REM !
   75 REM ld hl,Title
   80 REM Loop;ld a,(hl);!put let
ter in A register
   85 REM cp 0;! test end of mess
age
   90 REM jr z,Endit
   95 REM rst 16;! print it
  100 REM inc hl
  105 REM jr Loop
  110 REM Endit;pop hl
  115 REM pop af
  120 REM ret
  130 REM finish
```

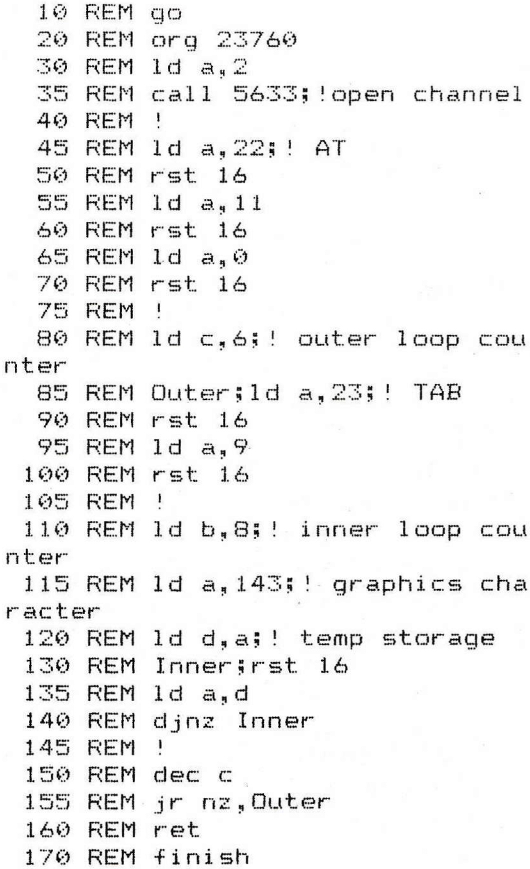
Figure 8.10

address before the routine is called; it then outputs characters from successive locations until it reaches a location containing a zero value. This subroutine can be called any time the output of text is required. The text must first be stored in memory using the DEFS instruction and this will be followed immediately with a DEFB instruction in order to store the zero value in memory.

8.6 Nested loops

We have looked at the way in which we can write instructions which are executed several times by putting them into a loop. There are many programs where a block of instructions, which includes a loop, is itself repeated several times. In other words, we have a loop included inside another loop. This type of structure is called a nested loop.

There are several different types of loop, any of which may be used as either the inside or outside loop. There is no limit to the type or number of loops which may be used in a nested loop structure. Figure 8.11 is a simple program which illustrates the programming of nested loops.



```
  10 REM go
  20 REM org 23760
  30 REM ld a,2
  35 REM call 5633;!open channel
  40 REM !
  45 REM ld a,22;! AT
  50 REM rst 16
  55 REM ld a,11
  60 REM rst 16
  65 REM ld a,0
  70 REM rst 16
  75 REM !
  80 REM ld c,6;! outer loop cou
nter
  85 REM Outer;ld a,23;! TAB
  90 REM rst 16
  95 REM ld a,9
 100 REM rst 16
 105 REM !
 110 REM ld b,8;! inner loop cou
nter
 115 REM ld a,143;! graphics cha
racter
 120 REM ld d,a;! temp storage
 130 REM Inner;rst 16
 135 REM ld a,d
 140 REM djnz Inner
 145 REM !
 150 REM dec c
 155 REM jr nz,Outer
 160 REM ret
 170 REM finish
```

Figure 8.11

8.7 Program

Write a subroutine which clears the screen by displaying on the screen 24 lines each consisting of 32 space characters.

By modifying the above subroutine write a program which displays a full screen of every printable character and keeps it there until any key is pressed on the keyboard.

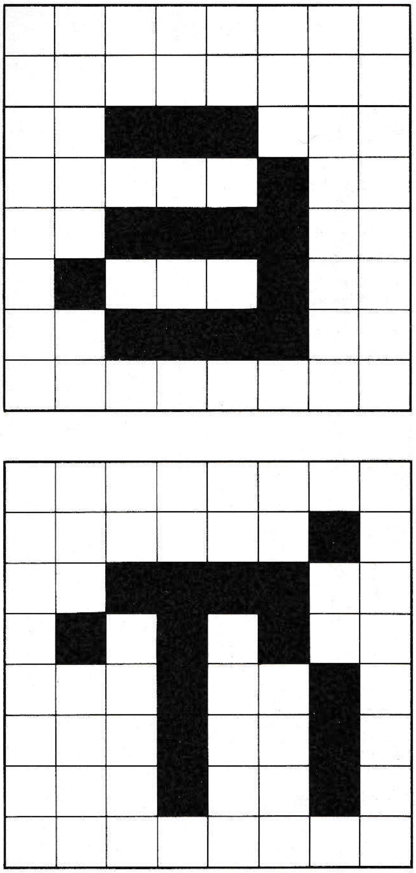
9 THE DISPLAY

9.1 The display file

Up to now all output to the display has been carried out using the character printing routine in the Spectrum ROM. We have been able to use this routine without considering how the Spectrum produces the display. To make maximum use of the display we need to look closer at how the display is produced.

The Spectrum has two areas of memory which are used in the production of the display. The second of these areas is the attribute file which controls the colours of the display; this will be described in Chapter 11. The main area used for the display is the display file which controls the shapes shown on the screen. In order to display anything on the screen it is necessary to load into the display file a series of numbers which specify the shape on the screen.

The screen of the Spectrum display has 24 lines of 32 characters, each character being made up from an 8×8 matrix of dots; Fig. 9.1 shows some examples of characters produced in this way. Each row of the character matrix is then stored as a one-byte number in the display file. The figure also shows this number, for each row of dots in the character matrix,in binary and decimal. Examining the binary numbers should show you how the number is produced, using a one to indicate the presence of a dot and a zero to show that there is no dot. Each character on the screen is stored as eight one-byte numbers, but they are not stored in successive memory locations. In the display file the screen is stored as three eight-line sections, so the first part of the display file stores the numbers which form the character matrices for the first eight lines of the display. Each section is stored as shown in Fig. 9.2. The 32 bytes which make up the top rows of the character matrix for the characters in the first line are stored in successive memory locations followed by the 32 bytes produced by the top rows of the characters in the second line. This is repeated until the top rows of the character matrices for the eight lines are stored. Now this process is repeated for the second and subsequent rows for the characters in the eight lines, as shown in Fig. 9.3.



0 0 0 0 0 0 0 0	0
0 0 0 0 0 0 0 0	0
0 0 1 1 1 0 0 0	56
0 0 0 0 0 1 0 0	4
0 0 1 1 1 1 0 0	60
0 1 0 0 0 1 0 0	68
0 0 1 1 1 1 0 0	60
0 0 0 0 0 0 0 0	0

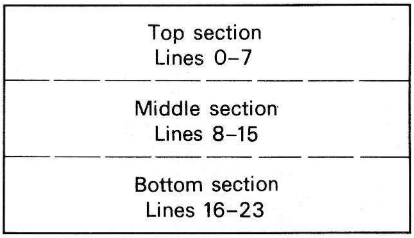
0 0 0 0 0 0 0 0	0
0 0 0 0 0 0 1 0	2
0 0 1 1 1 1 0 0	60
0 1 0 1 0 1 0 0	84
0 0 0 1 0 0 1 0	18
0 0 0 1 0 0 1 0	18
0 0 0 1 0 0 1 0	18
0 0 0 0 0 0 0 0	0

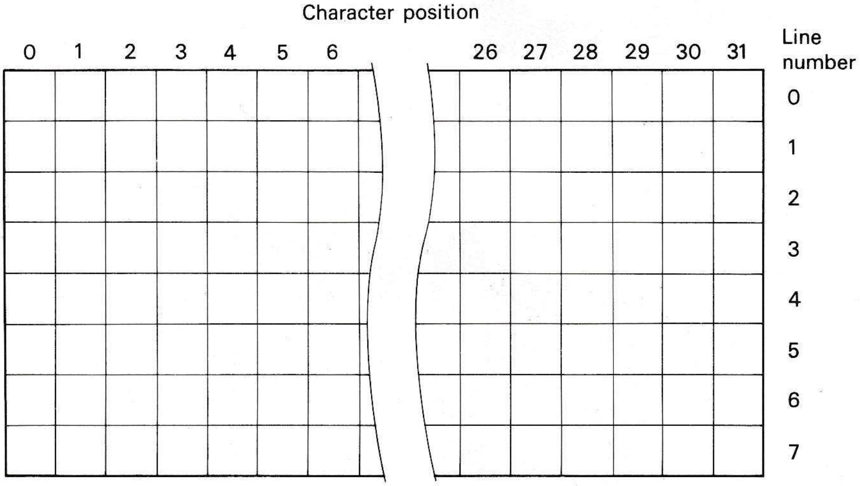
Figure 9.1

9.2 Finding the characters

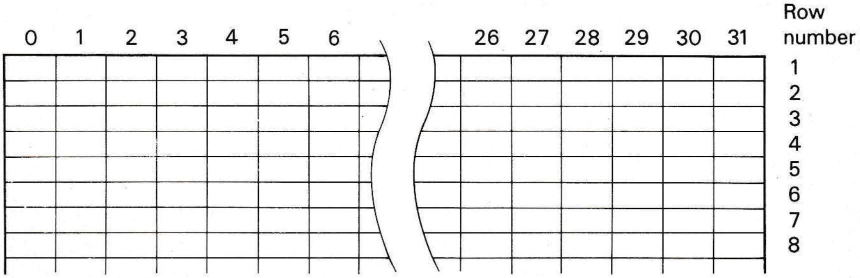
Each row of the character matrix is represented by an eight-bit binary number, which means that each row is stored in a single memory location. The numbers which make up the shapes of all the standard characters are stored in a block of memory, in the Spectrum ROM, starting at location 15 616. Each character is stored as eight numbers in successive memory locations. They are stored in character code order starting with a SPACE, which is ASCII code 32. This is then followed by the eight bytes used to make the shape of the character whose code is 33, and so on.

The location of the first byte for any character can be calculated from its character code using the formula:


Top section
Lines 0–7
Middle section
Lines 8–15
Bottom section
Lines 16–23


Character position
0 1 2 3 4 5 6 26 27 28 29 30 31
Line number
0
1
2
3
4
5
6
7

Top section of screen


0 1 2 3 4 5 6 26 27 28 29 30 31
Row number
1
2
3
4
5
6
7
8

One line of screen

Figure 9.2

(Character code − 32) * 8 + 15616

User-defined graphics characters are stored in the same way except that they are stored in RAM memory rather than ROM memory; this enables them to be changed by the programmer.

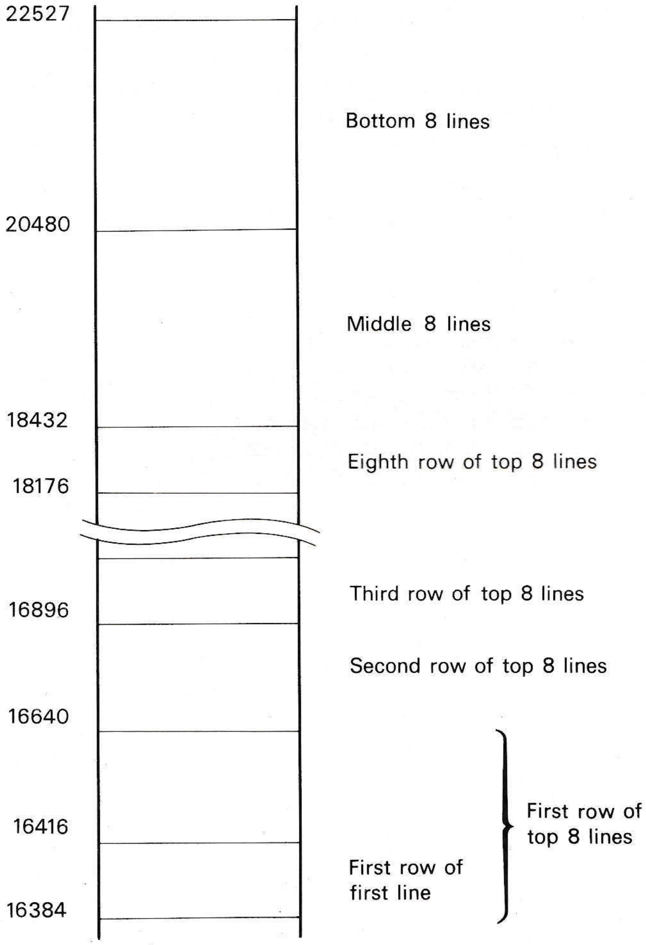

22527
Bottom 8 lines
20480
Middle 8 lines
18432
Eighth row of top 8 lines
18176
Third row of top 8 lines
16896
Second row of top 8 lines
16640
First row of top 8 lines
16416
First row of first line
16384

Figure 9.3

They are normally stored at the top of memory, starting at location 32 600 on the 16K machine and location 65 368 on the 48K machine. One of the advantages of using assembly language or machine code is that the programmer can set up as many user-defined characters as necessary in any area of unused RAM memory.

9.3 Block moves

All of the assembly language instructions we have seen so far use one or two items of data, but the Z80 microprocessor in the

Spectrum has several instructions which allow complete blocks of memory to be used.

The first group of instructions allow the contents of one block of memory to be moved to another block of memory. Remembering that the display file is stored in memory, obviously these instructions can be used for moving the display, or part of the display. As an example, the program segment in Fig. 9.4 moves

```
   10 REM go
   20 REM org 23760
   30 REM equ 16416 Line2
   40 REM equ 18464 Line10
   45 REM Temp;defw 0
   50 REM !initial values
   55 REM ld hl,Line10
   60 REM ld de,Line2
   65 REM ld c,8;!rows/line count
   70 REM Loopa;ld b,32;!bytes/li
ne
   75 REM Loopb;ld a,(hl)
   80 REM ld (de),a;!move a byte
   85 REM inc hl
   90 REM inc de
   95 REM djnz Loopb
  100 REM push de;!temp storage
  105 REM ld de,224
  110 REM add hl,de
  115 REM ld (Temp),hl
  120 REM pop hl
  125 REM add hl,de
  130 REM ex de,hl
  135 REM ld hl,(Temp)
  140 REM dec c
  145 REM jr nz,Loopa
  150 REM ret
  160 REM finish
```

Figure 9.4

the tenth line of the display to the second line. The important instruction is:

LDIR

which stands for load, increment, and repeat. Before the instruction is executed the HL register pair must contain the first address of the block which is to be moved, the DE register pair holds the address of the first location of the memory block to which it is to be moved, and the BC register pair holds the number of bytes of data to be moved. Because the number of bytes of data is held in a register pair, the number of bytes which can be moved is 65 535. The action of the LDIR instruction is to move the contents of one memory location, pointed to by the HL register pair, to the memory location pointed to by the DE register pair. The HL and DE register pairs are then increased by one and the BC register pair is decreased by one; this is repeated until the BC register pair is zero.

Using the LDIR instruction can cause problems if there is an overlap between the blocks of memory. Consider the following program segment which is an attempt to move the contents of one block to a second block with higher address:

```
LD HL,30000
LD DE,30100
LD BC,500
LDIR
```

When this segment is run the first one hundred bytes of the original block will be copied to the new block, but when the program reaches the second hundred bytes to be copied they have already been overwritten. The effect of this program segment will be to produce five copies of the first one hundred bytes of the original block.

The instruction which overcomes this problem is:

```
LDDR
```

which stands for load, decrement, and repeat. LDDR works in exactly the same way as the LDIR instruction except that at each step the contents of the HL and DE register pairs are decreased by one. Before the instruction is run, the HL and DE register pairs must point to the highest addresses in the two memory blocks.

There are two further instructions which allow blocks of memory to be copied. They are:

LDI and LDD

They are similar to the LDIR and LDDR instructions in that they copy a byte of data from one memory block to the other, the values in the HL and DE register pairs are changed to point to the next respective positions in the blocks, and the BC register pair is decreased by one. Unlike the previous instructions, they do not automatically repeat until the BC register pair contains zero. After each byte has been copied, a test for the end of the blocks must be included in the program and a jump back to the LDI or LDD instruction must be made if it is not complete. The LDI and

LDD instructions are used when the number of items of data to be moved is not known and the program will test for the end of the block.

If you are testing for a zero value in the BC register pair, this is shown, not by the zero flag but by the parity/overflow flag. The flag is set to zero if BC is zero; otherwise it is set to one. To test if the flag is zero the condition to be tested is PO and for one the condition is PE.

9.4 Some display routines

In this section I would just like to show you some relatively simple routines to enable you to scroll the display in a variety of ways. You will find that you do not yet understand all the instructions in the routines; do not worry, these will be explained shortly.

The first routine scrolls the whole screen to the left. The leftmost character in each line is moved to the end of the line above and the first character on the screen is lost. This is shown in Fig. 9.5.

```
   10 REM go
   20 REM org 23760
   30 REM equ 16384 Start
   40 REM equ 16385 Next
   55 REM ! initial values
   60 REM ld de,Start
   65 REM ld hl,Next
   70 REM ld b,192
   75 REM !
   80 REM Loop;push bc
   85 REM ld bc,31;!no. of bytes
   90 REM ld a,(de);!save 1st byt
e
   95 REM ldir;! move a row
  100 REM dec hl
  105 REM ld (hl),a;!wraparound
  110 REM !do next row
  115 REM inc hl
  120 REM inc hl
  125 REM inc de
  130 REM pop bc
  135 REM djnz Loop
  140 REM ret
  150 REM finish
```

Figure 9.5

The next routine also scrolls the whole screen, only this time it is scrolled down. The display is moved down one line at a time, shown in Fig. 9.6.

```
  10 REM go
  20 REM org 23760
  30 REM Temp;defw 0
  35 REM ld hl,22527;!bottom sec
tion
  40 REM call Secsc
  45 REM call Linem
  50 REM ld hl,20481;!middle sec
tion
  55 REM call Secsc
  60 REM call Linem
  65 REM ld hl,18431;!top sectio
n
  70 REM call Secsc
  75 REM call Blankl
  80 REM ret;!end of main progra
m
  85 REM !
  90 REM Secsc;ld b,8;! lines
  95 REM Loopa;push bc
 100 REM push hl
 105 REM ld bc,224
 110 REM ld de,32
 115 REM scf
 120 REM ccf
 125 REM sbc hl,de
 130 REM pop de
 135 REM lddr;!move section
 140 REM ld (Temp),hl
 145 REM ex de,hl
 150 REM scf
 155 REM ccf
 160 REM ld de,32
 165 REM sbc hl,de
 170 REM push hl
 175 REM ld hl,(Temp)
 180 REM scf
 185 REM ccf
 190 REM sbc hl,de
 195 REM pop de
 200 REM pop bc
 205 REM djnz Loopa
 210 REM ret
 215 REM !
 220 REM Linem;ld b,8
 225 REM Lineloop;push bc
 230 REM ld hl,1824
 235 REM add hl,de
 240 REM ld bc,32
 245 REM lddr;!move to next sect
ion
 250 REM ld bc,32
```

```
255 REM ex de,hl
260 REM scf
265 REM ccf
270 REM sbc hl,bc
275 REM ex de,hl
280 REM scf
285 REM ccf
290 REM sbc hl,bc
295 REM pop bc;!recover counter
300 REM djnz Lineloop
305 REM ret
310 REM !
315 REM Blankl;ld b,8
320 REM ld hl,16384
325 REM ld a,0
330 REM Loopd;push bc
335 REM ld b,32
340 REM Loopc;ld (hl),a
345 REM inc hl
350 REM djnz Loopc
355 REM ld de,224
360 REM add hl,de
365 REM pop bc
370 REM djnz Loopd
375 REM ret
380 REM finish
```

Figure 9.6

Finally, a useful display routine from the Spectrum ROM is the clear screen routine, which is achieved with the program segment shown in Fig. 9.7.

```
10 REM go
20 REM org 23760
30 REM ld b,24;!clear 24 lines
40 REM call 3652;!ROM routine
50 REM ret
60 REM finish
```

Figure 9.7

9.5 The screen border

The colour of the screen border can be changed very simply at any time by using the instruction:

OUT (254),A

where the A register contains the value of the required border colour.

The OUT instruction is used whenever the central processor of the computer sends data to the outside world. Any device which is connected to the computer either for the input or output of data is allocated a port and each port is numbered. As you can see, port 254 controls the colour of the border but this port is also used to control the loudspeaker. You may remember that this same port is also used for input from the keyboard.

Although the above instruction will change the colour of the border, the change will only be temporary unless the value of the new border colour is also stored in bits three to five of the system variable BORDER. The remaining bits of this system variable control the attributes of the bottom part of the screen. Appendix G includes details of a subroutine to change the border colour.

9.6 Program

Many interesting effects can be obtained by direct manipulation of the display file. Write a program which will invert the centre eight lines of the display. It will be necessary to use another area of memory as a temporary storage area for some of the display.

10 MULTIPLICATION AND DIVISION

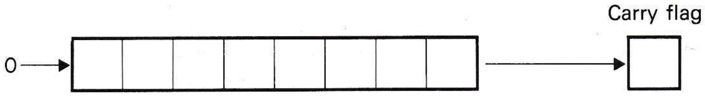
10.1 Shift instructions

Although we have already seen simple methods of multiplying and dividing using the ADD and SUB instructions, this chapter will discuss the instructions and the methods which will allow more efficient processing. The instructions required to carry out multiplication and division are called shifts.

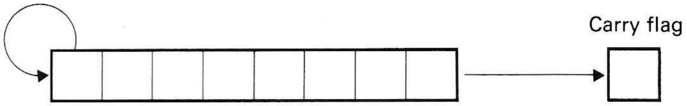
Shift instructions move the bits in a register or memory location one place to the left or the right. The Z80 microprocessor in the Spectrum has three shift instructions, SRL, SRA, and SLA. The SRL instruction which has the format:

SRL m

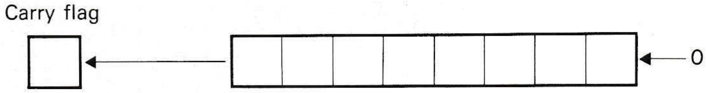
means 'shift right logical' and m can be any of the single eight-bit registers, or the contents of a memory location pointed to by the HL register pair. The logical right shift treats the value to be processed as a pattern of bits, all of the bits being moved one place to the right; a zero is moved into the end bit on the left and the original right end bit is moved into the carry flag. Figure 10.1 shows the operation of all the shift instructions.



Shift Right Logical (SRL)



Shift Right Arithmetic (SRA)



Shift Left Arithmetic (SLA)

Figure 10.1

The bit which is moved out is put into the carry flag by all of the shift instructions; conditional jump instructions can then be used to test the value of this bit.

The second of the shift instructions, the SRA instruction, has the format:

SRA m

which means 'shift right arithmetic' and m is again any single eight-bit register or memory location. The effect of the arithmetic right shift is the same as the logical right shift except that the value in the left end bit stays the same. An arithmetic right shift is the same as dividing the value in the register, or memory location, by two, the remainder being in the carry flag.

The final shift instruction is the SLA instruction, which has the format:

SLA m

which means 'shift left arithmetic' and m is the same as in the previous instructions. The shift left arithmetic moves all the bits to the left and moves a zero into the right end bit. The shift left instruction has the effect of multiplying the value in the register, or memory location, by two.

A logical left shift would be exactly the same as an arithmetic left shift so there is no need for a separate logical left shift instruction.

10.2 Multiplication

The method used for multiplication is known as 'shift and add'. The method is based on the same principles as long multiplication and is best illustrated by working through a multiplication in binary; but first let us work through a decimal multiplication to remind ourselves of the method:

```
     1537     Multiplicand
   × 2054     Multiplier
     ----
     6148     × 4
    7685      × 5
   0000       × 0
  3074        × 2
  -------------
  3156998     Product
```

Long multiplication in binary is exactly the same as in decimal except that you need only to be able to multiply by one or zero. So you can see it is much easier to do.

10101	
× 11001	
10101	× 1
00000	× 0
00000	× 0
10101	× 1
10101	× 1
1000001101	Product

The method of shift and add can be stated as follows:

Working from the right of the multiplier add the multiplicand to the product if the present bit of the multiplier is a one. Shift the multiplicand one bit to the left and repeat for the next bit of the multiplier until the end of the multiplier.

Figure 10.2 is a program which multiplies the contents of the B and C registers and leaves the product in the accumulator.

```
  10 REM go
  20 REM org 23760
  30 REM !Shift and add mutiplic
ation
  35 REM ld a,0
  40 REM ld b,7
  45 REM Loop;srl e
  50 REM !test if bit is one
  55 REM jp nc,Next
  60 REM add a,d;!add it to prod
uct
  65 REM Next;sla d
  70 REM djnz Loop
  75 REM ret
  80 REM finish
```

Figure 10.2

A method used for division is based on long division and is very similar to the 'shift and add' method. It is called 'shift and subtract'.

10.3 Rotations

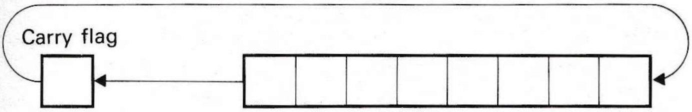
A group of instructions which are very similar in operation to the shift instructions are the rotate instructions. The main difference between shift and rotate instructions is that on the rotate the bit which is moved out is carried round and put in the other end. Like shifts, rotates move the bits one place to the left or to the right.

Some of these rotate instructions refer to the accumulator as part of the instruction and not as a separate operand. These instructions are one-byte instructions while the remainder of the rotate instructions, like the shift instructions, need to have the register, or memory location, specified; such instructions are two bytes long.

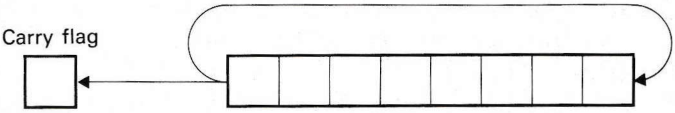
The accumulator rotations may be either to the right or to the left and may, or may not, include the carry flag in the rotation. The four instructions are:

RLA, RRA, RLCA, and RRCA

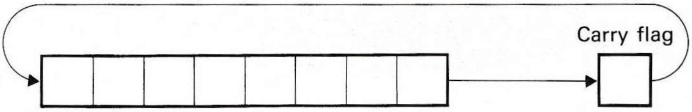
The operation of these instructions is shown in Fig. 10.3.



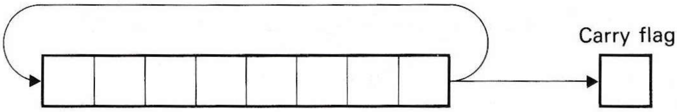
Rotate Left Accumulator (RLA)



Rotate Left Circular Accumulator (RLCA)



Rotate Right Accumulator (RRA)



Rotate Right Circular Accumulator (RRCA)

Figure 10.3

The rotate instructions for the other eight-bit registers or contents of a memory location pointed to by the HL register pair are:

RL m	Rotate left through the carry flag
RR m	Rotate right through the carry flag
RLC m	Rotate left circular, not through the carry
RRC m	Rotate right circular, not through the carry

where m is any of the eight-bit registers, except the accumulator, or a memory value pointed to by the HL pair.

The operation of these instructions is exactly as shown for the corresponding accumulator rotations.

10.4 Program

The multiplication program given earlier only deals with positive numbers; it should be relatively easy to extend it to deal with signed numbers. One method is to first multiply the positive values of the two numbers together and then use the rules of multiplication to determine the sign. The rule is that signs which are the same give a positive result and signs which are different give a negative value. Remember a signed number with a one in the left end bit is negative. Its positive value can be found using the NEG instruction.

Write a subroutine which carries out division by 10; then use this subroutine to write a further subroutine which will output the value in the accumulator as a number in the range −128 to +127. Negative numbers should be preceded by a '−' sign.

Using the above subroutine and your modified multiplication program, write a program which will input two signed numbers and calculate their product. A typical display would be:

```
4*-6=-24
  7*8=56
```

You may wish to try a similar program for division, but because you are working with integers you will probably be left with a remainder, so you should output the result and the remainder.

11 BITS OF LOGIC

11.1 Bit operations

Up till now, all of the instructions we have seen have used data in bytes. We have not dealt with any instructions using less than eight bits of data. Now we will look at some instructions which use one bit of data at a time. First we need to number the bits in a register, or memory location, so that we can refer to them. Figure 11.1 shows that the bits are numbered from right to left, starting from zero. I do not know why this is done but it is standard with all computers.

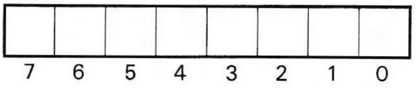


Figure 11.1

All the bit instructions operate on any of the eight-bit registers or the contents of a memory location pointed to by the HL register pair. The first of the instructions has the format:

```
BIT n,m
```

where n is the bit number and m is an eight-bit register, or memory location. The purpose of this instruction is to test the value in the stated bit and set the zero flag in accordance with the result. As the bit instruction is always followed by a conditional jump instruction, a typical program segment would be:

```
BIT 4,D
JP Z,NEXTPART
```

if register D contains the value 45 (00101101binary) bit 4, remember from the right starting from bit zero, would be zero and the program would jump to the instruction labelled NEXTPART.

In the display file all the pixels which make up the television picture are stored as bits and the BIT instruction can be used to test whether a particular pixel on the screen is on or off.

The other instructions which act on particular bits are used to set the value of the bit to either one or zero. The SET instruction is used to give the value 1 to a particular bit. The instruction which gives a bit the value of 0 is the RES instruction. The format of the two instructions is:

SET n,m and RES n,m

where n is the number of the bit and m is the register, or memory location. Now that we have the capability of working with single bits we can look at a routine which gives a smooth side scroll of the screen by scrolling one bit at a time. Figure 11.2 is a routine to scroll the screen to the left. You should be able to modify this routine to scroll to the right.

```
 10 REM go
 20 REM org 23760
 30 REM equ 16384 Dispfile
 40 REM !
 45 REM ld hl,Dispfile
 50 REM ld c,192
 55 REM ! Main program
 60 REM Nextrow;ld a,(hl)
 65 REM sla a;!move 1st byte
 70 REM ld (hl),a
 75 REM !rest of row
 80 REM ld b,31
 85 REM Linemove;inc hl
 90 REM ld a,(hl)
 95 REM sla a;!scroll byte left
100 REM ld (hl),a
105 REM !test if 1st bit was 1
110 REM jp nc,Nocarry
115 REM dec hl;!previous byte
120 REM ld a,(hl)
125 REM set 0,a;!make bit 1
130 REM ld (hl),a
135 REM inc hl
140 REM Nocarry;djnz Linemove
145 REM inc hl
150 REM dec c
155 REM jr nz,Nextrow
160 REM ret
170 REM finish
```

Figure 11.2

11.2 Logic instructions

The Spectrum assembly language has a group of instructions which allow the bits in the accumulator to be combined with the bits in an eight-bit register or memory location according to the rules of the logical operators.

The logical operators work on corresponding bits in the accumulator and the operand. The basic logic operators are AND, OR, XOR, and NOT. The rules for combining bits are shown in Fig. 11.3. Apart from the NOT operator, all of the operations combine corresponding pairs of bits and produce a one-bit result for each pair.

AND operator

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

OR operator

0 AND 0 = 0
0 AND 1 = 1
1 AND 0 = 1
1 AND 1 = 1

XOR operator

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

NOT operator

NOT 0 = 1
NOT 1 = 0

Figure 11.3

The logic instructions are:

AND OR XOR

and the format of the instructions is:

AND p

where p is an eight-bit value either as a number, an eight-bit register, or the value in a memory location pointed to by the HL register pair. Figure 11.4 gives some examples of the working of the logic instructions. When a logic instruction is executed the sign and zero flags are set according to the value remaining in the accumulator.

The NOT operation is performed by the instruction CPL. The

Contents of A	01011100
Contents of B	11001100
Contents of A after AND B	00001100
Sign flag = 0	Zero flag = 0
Contents of A	01011100
Contents of B	11001100
Contents of A after OR B	11011100
Sign flag = 1	Zero flag = 0
Contents of A	01011100
Contents of B	11001100
Contents of A after XOR B	10010000
Sign flag = 1	Zero flag = 0
Contents of A	01011100
Contents of A after CPL	10100011

Figure 11.4

CPL instruction works only on the accumulator and its effect is to change the value of all the bits in the accumulator; all the ones are changed to zeros and all the zeros are changed to ones.

11.3 Packed data

You, as a programmer, should always be trying to minimize the amount of memory that is used by your programs and your data. The less money you can use, the more space you will have left for either a more comprehensive program or for more data. I was recently shown a mailing list program for the 16K ZX81. It was a program that had all the facilities you could possibly require from such a program but it had one major drawback: it was too big. When the program was in memory it only left enough space for 10 to 12 names and addresses.

Using assembly language will help to reduce the size of your program, but how can we reduce the amount of space taken by the data? Often the range of the data we need is much less than the range we can hold in a single memory location. In a single memory location we can hold a number with a range from 0 to 255 or from −128 to +127. Now suppose one of our data items is a person's age; obviously we do not need negative numbers but even the range 0 to 255 is much wider than necessary. If we use just seven of the eight bits for the age we can cover a range from 0 to 127, which is easily enough for any application, and we are left with the eighth bit which can be used to hold some other item of data. In fact, in one bit we can hold any data item which can have only two values, such as whether a person is male or female, whether they are members or non-members, or whether they are customers or suppliers. Any of these data items could be stored as the eighth bit in the memory location which holds the person's age. This is known as packing of data.

Figure 11.5 shows a typical example from a club membership list, where because the age range of members is 18 to 65, bits 0 to 5 are used to hold the age as the number of years over 18, bit 6 is used to hold the sex of the member, and bit 7 is used to show whether membership subscription has been paid or not. The example shows a paid-up woman of 45. In a club with a membership of 1000, using this method will only use 1000 bytes of memory, instead of 3000 bytes if each data item had been stored in a separate memory location.

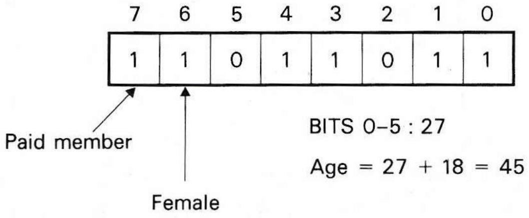


Figure 11.5

11.4 Packing and unpacking data

We have looked at the principle of packing data into a byte; now we can see how the data are put together and then how we can separate them again. Generally, putting data together uses the shift and OR instructions, and separating the data uses the shift and AND instructions.

Figure 11.6 is a program to pack data into the memory location labelled ITEM using the rules gives in the previous section. At the

```
  10 REM go
  20 REM org 23760
  30 REM Item;defb 0
  35 REM ld a,b
  40 REM sub 18;! age in bits 0-
5
  45 REM !
  50 REM ld b,6
  55 REM Sex;sla c;! move Sex to
 bit 6
  60 REM djnz Sex
  65 REM or c;!add sex to A
  70 REM !
  75 REM ld b,7
  80 REM Paid;sla d;!move Paid t
o bit 7
  85 REM djnz Paid
  90 REM or d;!add paid to A
  95 REM !
 100 REM ld (Item),a;!save in me
mory
 105 REM ret
 110 REM finish
```

Figure 11.6

start of the program the B register holds the actual age of the person, the C register the sex, and the D register whether they are paid-up or not.

This is followed by Fig. 11.7 which is a program for unpacking the data. This program reverses the procedure of the previous program.

```
  10 REM go
  20 REM org 23760
  30 REM !unpack data in ITEM
  35 REM ld a,(Item)
  40 REM and $80;!10000000B
  45 REM ld d,a
  50 REM ld b,7
  55 REM Paid1;srl d;!move to bi
t 0
  60 REM djnz Paid1
  65 REM !
  70 REM ld a,(Item)
  75 REM and $40;!01000000B
  80 REM ld c,a
  85 REM ld b,6
  90 REM Sex1;srl c;!move to bit
 0
  95 REM djnz Sex1
 100 REM !
 105 REM ld a,(Item)
 110 REM and $3F;!00111111B
 115 REM add a,18;!correct age
 120 REM ld b,a
 125 REM ret
 130 REM finish
```

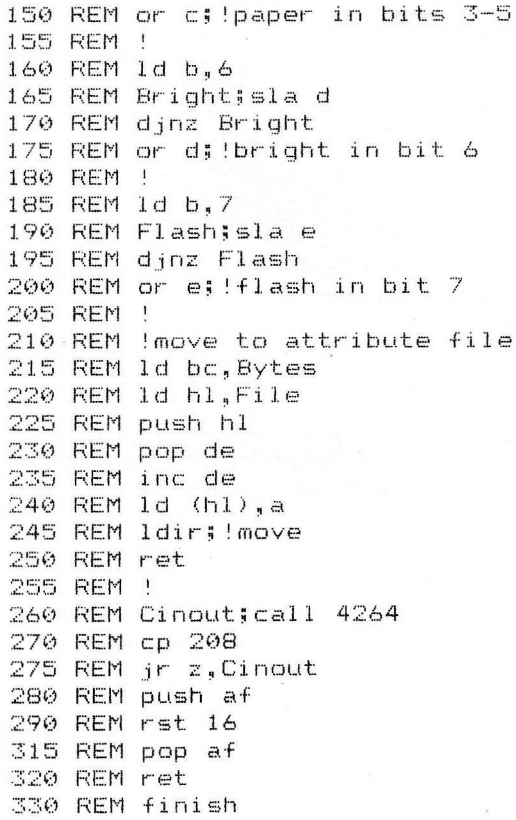
Figure 11.7

11.5 The attribute file

We have already discussed how the shapes seen on the television screen are stored as pixels in the display file, but the Spectrum is a colour computer and there is a further area of the memory used to store the colour details. This is the attribute file. The attribute file consists of one memory location for each character position on the screen and they are stored in the way you would expect them to be stored: first 32 locations for the top line, then 32 for the second line, and so on for all 24 lines. Each location in the attribute file is actually packed with four data items. Bits 0 to 2 hold the code for the ink colour in binary, bits 3 to 5 hold the paper colour, bit 6 is 1 for bright and 0 for normal, and bit 7 is 1 for flashing and 0 for steady.

Figure 11.8 shows how the attribute file can be used to change the colour characteristics of the screen.

```
  10 REM go
  20 REM org 23760
  24 REM equ 768 Bytes
  26 REM equ 22528 File
  30 REM !change colour items in
attribute file
  35 REM !input ink colour
  40 REM call Cinout
  45 REM sub 48;!code to value
  50 REM ld b,a
  55 REM !
  57 REM !input paper colour
  60 REM call Cinout
  65 REM sub 48
  70 REM ld c,a
  75 REM !
  77 REM !input bright?
  80 REM call Cinout
  85 REM sub 48
  90 REM ld d,a
  95 REM !
  98 REM !input flash?
 100 REM call Cinout
 105 REM sub 48
 110 REM ld e,a
 115 REM !
 120 REM !pack data in A
 125 REM xor a;!zero in A
 130 REM ld a,b;!ink in bits 0-2
 132 REM !
 135 REM ld b,3
 140 REM Paper;sla c
 145 REM djnz Paper
```



```
150 REM or c;!paper in bits 3-5
155 REM !
160 REM ld b,6
165 REM Bright;sla d
170 REM djnz Bright
175 REM or d;!bright in bit 6
180 REM !
185 REM ld b,7
190 REM Flash;sla e
195 REM djnz Flash
200 REM or e;!flash in bit 7
205 REM !
210 REM !move to attribute file
215 REM ld bc,Bytes
220 REM ld hl,File
225 REM push hl
230 REM pop de
235 REM inc de
240 REM ld (hl),a
245 REM ldir;!move
250 REM ret
255 REM !
260 REM Cinout;call 4264
270 REM cp 208
275 REM jr z,Cinout
280 REM push af
290 REM rst 16
315 REM pop af
320 REM ret
330 REM finish
```

Figure 11.8

11.6 Program

Write a subroutine which will output the value in the accumulator to the screen in binary. You will need to look at each bit separately and output the character code for 1 or 0 depending on the value of the bit.

Write a program which will input two decimal numbers, in the range 0 to 255, and a logic operator, AND, OR, or XOR, and will then output the two numbers in binary and also the result of combining them according to the given logic operator, in binary.

12 BLOCKS AND TABLES

12.1 Block searches

In addition to the instructions which enable complete blocks of memory to be moved we will now look at instructions which allow blocks of memory to be searched for a particular value. Before any of the block search instructions are given the accumulator must hold the value for which the computer is searching. The HL register pair holds the address of the first location to be searched and the BC register pair holds the number of memory locations which could be searched.

The first of the instructions is CPIR, which stands for 'compare, increment, and repeat'. This searches through the block of memory starting with the location pointed to by the HL register pair. After the value in each location has been compared with the accumulator the value in the HL register pair is increased by one and the value in the BC register is decreased by one. The search continues until either a match is found or the value in the BC register becomes zero. When a match is found the zero flag is set—remember that by becoming zero the BC register does not set the zero flag.

Similar to the last instruction is the CPDR instruction. This is the 'compare, decrement, and repeat' instruction and works in the same way as CPIR except that the HL register pair must initially point to the highest address of the block. When the instruction is executed the value in the HL register pair is decreased. Figure 12.1 is a short program which searches a block of memory until it finds the value 255 in a memory location.

```
  10 REM go
  20 REM org 23760
  30 REM equ 32600 Block
  35 REM Find;defw 0
  40 REM ld hl,Block
  45 REM ld bc,100;!no. of locat
ions
  50 REM ld a,255;!value searche
d
  55 REM cpir;! search
  57 REM !jump if not found
  60 REM jr nz,Endit
  65 REM ld (Find),hl;!location
of value
  70 REM Endit;ret
  75 REM finish
```

Figure 12.1

Finally there are two block search instructions which do not repeat automatically; they are CPI and CPD. These are very similar to the two previous instructions except for the repeat. After each comparison the programmer must write extra instructions to test whether a match between the accumulator and the value in memory has been found; if no match is found, a test must be carried out to determine if the end of the block has been reached. These instructions would be used when further processing is required.

12.2 Index registers

We have seen that the HL register pair is often used as a pointer, to enable data to be used directly from memory. There are two other 16-bit registers which are used as pointers to data in memory; they are the IX and IY registers. The IX and IY registers are known as index registers and are normally used in conjunction with blocks of memory.

Unfortunately the Spectrum uses the IY register as a pointer to the block of memory holding the system variables, so it is not available to the assembly language programmer.

In use, an index register is used to point to the first location in a block of memory and the other locations in the block are referred to by their displacement from the start of the block. The program segment in Fig. 12.2 shows how an index register is used and how

```
   10 REM go
   20 REM org 23760
  100 REM equ 32600 Block
  110 REM !
  120 REM ld ix,Block
  125 REM !
  130 REM !
  140 REM set 5,(ix+2);!3rd locat
ion
  145 REM !
  150 REM !
  160 REM dec (ix+4);!5th locatio
n
  165 REM !
```

```
 170 REM !
 180 REM ld a,(ix+8);!9th locati
on
 185 REM !
 190 REM !
 200 REM finish
```

Figure 12.2

particular locations are referred to by specifying the index register plus the displacement from the start of the block.

The index registers are normally used when reference needs to be made to a block of related data such as a table. The IX index register can still be used even if it is not necessary to refer to a block of data by using a zero displacement to refer to the location pointed to by the index register. An index register can be used in place of the HL pair in any instruction which uses the HL as a pointer to a memory location, but it must be followed by a displacement.

12.3 Look-up tables

Very often, related data are stored in the computer as a table or list. This can then be searched to see if it contains a given item of data. If the data item is found in a particular list or table, this is used to trigger a jump to a new part of the program. Of course, if the data are in another table the jump will be to another part of the program. A program using look-up tables can either consist of several different tables, each with a related jump, or of a single table with a different jump for each entry in the table. The program in Fig.12.3 is an example of the first type. It uses several different tables and finding an entry in a particular table causes a jump to a new part of the program. Each table has its own related jump.

```
  10 REM go
  15 REM org 23760
  20 REM !Single step program
  25 REM !
  30 REM !space for tables
  35 REM Twobyte;defb 6 14 16 22
 24 30 32 38 40 46 48 54 56 63 1
98 203 206 211 214 219 222 230 2
38 246 254
  40 REM Threebyte;defb 1 17 33
34 42 49 50 58 194 195 196 202 2
04 205 210 212 218 220 226 228 2
34 242 244 250 252
  45 REM Indtwo;defb 9 25 35 41
```

```
43 57 225 227 229 233
  50 REM Indfour;defb 33 42 34 2
03
  55 REM Edbyte;defb 34 42 67 75
 83 91 115 123
  60 REM !Running area for an in
struction
  65 REM Instloc;defb 0 0 0 0 20
1
  80 REM Progcount;defw 0
  85 REM !save registers for MC
program
  90 REM ld hl,0
  95 REM push hl
 100 REM push hl
 105 REM push hl
 110 REM push hl
 115 REM !input start address
 120 REM call Numinput
 125 REM ld (Progcount),hl
 130 REM ex de,hl;!DE is counter
 135 REM !find no. of bytes/inst
ruction
 140 REM Start;ld hl,Instloc
 145 REM ld a,(de)
 150 REM !test if end of MC prog
 155 REM cp 201;! RET instructio
n
 160 REM ret z
 165 REM !load 1st byte
 170 REM ld (hl),a
 175 REM !index instruction?
 180 REM cp 221;!IX
 185 REM jp z,Indexcode
 190 REM cp 253;!IY
 195 REM jp z,Indexcode
 200 REM !"237" instruction
 205 REM cp 237
 210 REM jp z,Edcode
 215 REM !2 byte instruction?
 220 REM ld hl,Twobyte
 225 REM ld b,25;!length of tabl
e
 230 REM Loopa;cp (hl)
 235 REM jp z,Instwo
 240 REM inc hl
 245 REM djnz Loopa
 250 REM !3 byte instruction
 255 REM ld hl,Threebyte
 260 REM ld b,26;!length of tabl
e
 265 REM jp Run
 270 REM jp z,Insthree
 275 REM inc hl
 280 REM djnz Loopb
 285 REM !1 byte instruction
 290 REM jp Run
 295 REM !index instructions
 300 REM Indexcode;inc de;!2nd b
yte
 305 REM ld a,(de)
 310 REM ld hl,Indtwo
 315 REM ld b,10;!length of tabl
e
 320 REM Loopc;cp (hl)
 325 REM jp nz,Next
 330 REM ld hl,Instloc+1
 335 REM ld (hl),a
 340 REM jp Run
 345 REM !continue search
 350 REM Next;inc hl
 355 REM djnz Loopc
 360 REM !4 byte instruction?
 365 REM ld hl,Indfour
 370 REM ld b,4;!length of table
 375 REM Loopd;cp (hl)
 380 REM jp z,Instfour
 385 REM inc hl
 390 REM djnz Loopd
 395 REM !save 3 byte inst
 400 REM ld hl,Insloc+1
 405 REM ld (hl),a
 410 REM inc de
 415 REM inc hl
 420 REM ld a,(de)
 425 REM ld (hl),a
 430 REM jp Run
 435 REM !save 4 byte inst
 440 REM ld hl,Instloc+1
 445 REM ld (hl),a
 450 REM inc de
 455 REM inc hl
 460 REM ld a,(de)
 465 REM ld (hl),a
 470 REM inc de
 475 REM inc hl
 480 REM ld a,(de)
 485 REM ld (hl),a
 490 REM jp Run
 495 REM !"237" instruction
 500 REM Edcode;inc de
 505 REM ld hl,Edbyte
 510 REM ld b,8;!length of table
 515 REM ld a,(de)
```

```
520 REM Loope;cp (hl)
525 REM jp z,Instfour
530 REM inc hl
535 REM djnz Loope
540 REM !it is 2 byte inst
545 REM jp Instwoa
550 REM !2 byte instruction
555 REM Instwo;inc de
560 REM ld a,(de)
565 REM Instwoa;ld hl,Instloc+1
570 REM ld (hl),a
575 REM jp Run
580 REM !3 byte instruction
585 REM Insthree;inc de
590 REM ld hl,Instloc+1
595 REM ld a,(de)
600 REM ld (hl),a
605 REM inc de
610 REM inc hl;!3rd byte
615 REM ld a,(de)
620 REM ld(hl),a
625 REM jp Run
630 REM !save location of next
instruction
640 REM Run;inc de
650 REM ex de,hl
655 REM ld (Progcount),hl
660 REM !restore registers
665 REM pop hl
670 REM pop de
675 REM pop bc
680 REM pop af
685 REM !run the instruction
690 REM call Instloc
695 REM !display registers
700 REM call Clearscreen
705 REM call Headings
710 REM call Print
715 REM ld a,b
720 REM call Printa
725 REM ld a,c
730 REM call Print
735 REM ld a,d
740 REM call Printa
745 REM ld a,e
750 REM call Print
755 REM ld a,h
760 REM call Printa
765 REM ld a,l
770 REM call Print
775 REM push af
```

```
 780 REM pop hl
 785 REM ld a,l
 790 REM call Fprint
 795 REM !save registers
 800 REM push af
 805 REM push bc
 810 REM push de
 815 REM push hl
 820 REM ld hl,(Progcount)
 825 REM ex de,hl
 830 REM ld a,d
 835 REM call Printa
 840 REM ld a,e
 845 REM call Print
 850 REM !empty running space
 855 REM ld a,0
 860 REM ld b,4
 865 REM ld hl,Instloc
 870 REM Loopf;ld (hl),a
 875 REM inc hl
 880 REM djnz Loopf
 882 REM call Keyscan
 885 REM jp Start
 890 REM !subroutines
 895 REM !
 900 REM !clearscreen
 905 REM Clearscreen;push af;pus
h bc
 910 REM push de;push hl
 915 REM ld b,24
 920 REM call 3652
 925 REM pop hl;pop de
 930 REM pop bc;pop af
 935 REM ret
 940 REM !
 945 REM !print titles
 950 REM Acc;defs AccumulatorBCD
EHL S  Z  -  H  - P/V N  CFlagsP
C
 955 REM Headings;ld a,2
 960 REM call 5633
 965 REM ld a,13
 970 REM ld b,4
 975 REM Head1;ld c,a
 980 REM rst 16
 985 REM ld a,c
 990 REM djnz Head1
 995 REM !
1000 REM ld hl,Acc
1005 REM ld b,11
1010 REM Acchead;ld a,(hl);!Accu
mulator
```

```
1015 REM rst 16
1020 REM inc hl
1025 REM djnz Acchead
1030 REM call Newlines
1035 REM !
1040 REM ld b,3
1045 REM Registers;ld a,(hl)
1050 REM rst 16
1055 REM inc hl
1060 REM ld a,(hl)
1065 REM rst 16
1070 REM call Newlines
1075 REM inc hl
1080 REM djnz Registers
1085 REM !flag headings
1090 REM ld b,7
1095 REM ld a,32
1100 REM ld c,a
1105 REM Spaces;rst 16
1110 REM ld a,c
1115 REM djnz Spaces
1120 REM !
1125 REM ld b,23
1130 REM Flags;ld a,(hl)
1135 REM rst 16
1140 REM inc hl
1145 REM djnz Flags
1150 REM ld a,13
1155 REM rst 16
1160 REM ld b,5
1165 REM Flag;ld a,(hl)
1170 REM rst 16
1175 REM inc hl
1180 REM djnz Flag
1185 REM call Newlines
1190 REM ld a,(hl)
1195 REM rst 16
1200 REM inc hl
1205 REM ld a,(hl)
1210 REM rst 16
1215 REM pop hl;pop de;pop bc;po
p af
1220 REM ret
1225 REM !
1230 REM Newlines;ld a,13;rst 16
1235 REM ld a,13;rst 16
1240 REM ret
1245 REM !
1250 REM Printa;push af
1255 REM ld a,6;rst 16
1260 REM pop af
1263 REM !print value in hexadec
```

```
imal
1265 REM Print;push af;push bc;p
ush af
1270 REM and 240
1275 REM ld b,4
1280 REM Shift;srl a
1285 REM djnz Shift
1290 REM call Digit
1295 REM pop af
1300 REM and 15
1305 REM call Digit
1310 REM pop bc;pop af
1315 REM ret
1320 REM !
1325 REM !print flags
1330 REM Fprint;push af;push bc
1335 REM ld b,8
1340 REM Fleft;sla a
1345 REM jp nc,Fzero
1350 REM push af
1355 REM ld a,49
1360 REM jr Fout
1365 REM Fzero;push af
1370 REM ld a,48
1375 REM Fout;rst 16;pop af
1380 REM djnz Fleft
1385 REM pop bc;pop af
1390 REM ret
1400 REM !wait for key press
1405 REM Keyscan;in a.(254)
1410 REM cpl;jr z,Keyscan;ret
1415 REM !print hex digit
1420 REM Digit;cp 9;jp p,Letter
1425 REM add a,48;jr Outd
1430 REM Letter;add a,55
1435 REM Outd;rst 16;ret
1440 REM !number input
1445 REM Numinput;ld l,0;ld h,0
1450 REM Numloop;call Cinout
1455 REM cp 13;ret z
1460 REM call Multen;sub 48
1465 REM ld d,0;ld e,a;add hl,de
1470 REM jr Numloop
1475 REM !
1480 REM Multen;add hl,hl
1485 REM push hl;pop de;!copy to
 DE
1490 REM add hl,hl;add hl,hl
1495 REM add hl,de;ret
1500 REM !
1505 REM Cinout;push hl;push de
1510 REM push bc;Cinloop;call 42
```

```
64
1515 REM cp 208;jr z,Cinloop
1520 REM push af;rst 16
1525 REM pop af;pop bc;pop de;po
p hl
1530 REM ret
1550 REM finish
```

Figure 12.3

Besides illustrating the use of look-up tables this program forms a very useful utility program for the machine code or assembly language programmer. This is a single-step program which can be used to run another machine code program one instruction at a time. Initially the starting address of the machine code program is input and then one instruction is executed and the values in the registers are displayed. The next instruction is run when any key is pressed.

12.4 Jump tables

One very common programming requirement is to test the value of a variable and jump to a particular section of the program, depending on the value of the variable.

The method is to store a table of jump instructions in memory, using either the HL or IX registers as pointers to the start of the table. The value of the variable is used to calculate a displacement from the start of the jump table. This is added to the value in the pointer register to give the position in the jump table. The instructions:

JP (HL)   or   JP (IX)

are used to make the jump to the required part of the program. Figure 12.4 shows the use of this technique. The program inputs a month number, between 1 and 12, and from this displays the name of the month.

```
  10 REM go
  20 REM org 23760
  30 REM !Input month number,pri
nt name
  40 REM call Cinout
  45 REM sub 48;!code to value
  50 REM ld d,0;!set to 0
  55 REM ld e,a;!value in E
  60 REM sla a;!times 2
  65 REM add a,e;!times 3
  70 REM sub 3
```

```
 75 REM ld e,a
 80 REM ld hl,Jumtab
 85 REM add hl,de
 90 REM jp (hl);!jump to table
 95 REM !Jump table
100 REM Jumtab;jp Jan
105 REM jp Feb
110 REM jp Mar
115 REM jp Apr
120 REM jp May
125 REM jp Jun
130 REM jp Jul
135 REM jp Aug
140 REM jp Sep
145 REM jp Oct
150 REM jp Nov
155 REM jp Dec
160 REM !
165 REM Jan;ld hl,Ja
170 REM call Text
175 REM ret
180 REM !
185 REM Feb;ld hl,F
190 REM call Text
195 REM ret
200 REM !
205 REM Mar;ld hl,Mr
210 REM call Text
215 REM ret
220 REM !
225 REM Apr;ld hl,Ap
230 REM call Text
235 REM ret
240 REM !
245 REM May;ld hl,My
250 REM call Text
255 REM ret
260 REM !
265 REM Jun;ld hl,Jn
270 REM call Text
275 REM ret
280 REM !
285 REM Jul;ld hl,Jl
290 REM call Text
295 REM ret
300 REM !
305 REM Aug;ld hl,Au
310 REM call Text
315 REM ret
320 REM !
325 REM Sep;ld hl,S
330 REM call Text
```

```
335 REM ret
340 REM !
345 REM Oct;ld hl,O
350 REM call Text
355 REM ret
360 REM !
365 REM Nov;ld hl,N
370 REM call Text
375 REM ret
380 REM !
385 REM Dec;ld hl,D
390 REM call Text
395 REM ret
400 REM !
405 REM Ja;defs January
410 REM defb 0
415 REM F;defs February
420 REM defb 0
425 REM Mr;defs March
430 REM defb 0
435 REM Ap;defs April
440 REM defb 0
445 REM My;defs May
450 REM defb 0
455 REM Jn;defs June
460 REM defb 0
465 REM Jl;defs July
470 REM defb 0
475 REM Au;defs August
480 REM defb 0
485 REM S;defs September
490 REM defb 0
495 REM O;defs October
500 REM defb 0
505 REM N;defs November
510 REM defb 0
515 REM D;defs December
520 REM defb 0
525 REM !
530 REM Cinout;call 4264
535 REM cp 208;jr z,Cinout
540 REM push af;rst 16
555 REM pop af;ret
560 REM !
565 REM Text;ld a,2;call 5633
570 REM Rept;ld a,(hl)
575 REM cp 0
580 REM jr z,Endit
585 REM rst 16
590 REM jr Rept
595 REM Endit;ld a,13
600 REM rst 16
```

```
605 REM ret
610 REM finish
```

Figure 12.4

This is, of course, a very simple use of a jump table, because each program segment is the same length and it would be easier to calculate the start of each program segment.

12.5 Random numbers

For many games programs, it is essential to be able to produce random numbers in order to introduce into the game the necessary chance factor, but producing random numbers from an assembly language program can be a very difficult task.

The Spectrum ROM contains a very good random number generator; however, it is not simple to use and is only recommended to experienced programmers who have a good understanding of the ROM program. There is, of course, one easy way in which any assembly language program can use the random numbers generated by the ROM program. By returning to BASIC in the middle of your program, you can use the RND function to produce a random number, POKE the value into a suitable memory location, and finally return to your assembly language program with a USR call.

There is another way in which an assembly language program can produce a number which, although it is not really random, can under most circumstances be used as a random number. One of the registers in the Z80 microprocessor, called the R register, is used by the system to ensure that data are not lost from memory; in effect this means that the value in the R register is constantly changing and if the value is loaded into the accumulator using the instruction:

```
LD A,R
```

the accumulator will be loaded with a value between 0 and 255 which is reasonably random. If your program involves inputs from the keyboard, which, of course, means you are using loops of indeterminate length while waiting for a key to be pressed, this method probably gives numbers which are as random as the random number generator in the ROM.

Although by using the R register a number between 0 and 255 is given, it is not necessary to use all of this range. As an example let us see how this method can be used to simulate rolling a dice.

We need random numbers between 1 and 6, so if we take bits 0 to 2 of the number in the R register we will have a number in the range 0 to 7; by ignoring any zeros or sevens we are then left with a number in the required range. Figure 12.5 is a program which simulates the rolling of two dice until a key is pressed.

```
  10 REM go
  20 REM org 23760
  25 REM !1st die
  30 REM Start;ld a,r;!get numbe
r
  40 REM and 7;!00000111B
  45 REM cp 0;! 0 not in range
  50 REM jp z,Start
  55 REM cp 7;! 7 not in range
  60 REM jp z,Start
  65 REM add a,48;!value to code
  70 REM ld c,a;!temp store
  75 REM !2nd die
  80 REM Next;ld a,r
  85 REM and 7
  90 REM cp 0
  95 REM jp z,Next
 100 REM cp 7
 105 REM jp z,Next
 110 REM add a,48
 115 REM ld b,a
 120 REM !Print values
 125 REM ld a,2;call 5633;!open
channel
 130 REM !Print at 10,10
 135 REM ld a,22;!AT
 140 REM rst 16
 145 REM ld a,10;rst 16
 150 REM ld a,10;rst 16
 155 REM ld a,b;rst 16
 160 REM !Print at 10,13
 165 REM ld a,22;rst 16
 170 REM ld a,10;rst 16
 175 REM ld a,13;rst 16
 180 REM ld a,c;rst 16
 185 REM !test for key
 190 REM in a,(254)
 195 REM cpl;jp z,Start
 215 REM ret
 220 REM finish
```

Figure 12.5

12.6 Program

Write a program for a simple 'Guess the Number' game. The computer should produce a random number in the range 0 to 99 and then the player should be given five chances to try to guess the number. After each guess the player should receive a message to tell him or her whether the guess was too high or too low.

13 MORE ARITHMETIC

13.1 Sixteen-bit numbers

All of the numbers we have used for arithmetic have been eight-bit numbers, which means we have only been able to carry out arithmetic within a very limited range of numbers. However, the Spectrum has several 16-bit registers in its central processor, and the use of 16-bit numbers would give us a greatly increased range of numbers—enough for most problems. The Z80 microprocessor does include some 16-bit arithmetic instructions; indeed, these instructions can be used to provide 32-bit arithmetic, 48-bit arithmetic, or even longer numbers.

When carrying out 16-bit arithmetic, the HL register pair is used as an accumulator. The 16-bit addition instruction, which has the format:

```
ADD HL,ss
```

where ss is one of the 16-bit registers BC, DE, HL, or SP, adds the value in the register ss to the value in HL, and leaves the result in HL. As well as the straightforward ADD instruction, there is a second instruction for 16-bit addition; this is the ADC instruction which has the same format as the ADD instruction. The difference between ADD and ADC is that the ADC instruction, as well as adding the value in the register ss to the value in HL, also adds the value in the carry flag to HL. This means that a carry from a previous addition can be added in, so that 32-bit, or more, arithmetic can be performed. Figure 13.1 is a program which adds together two 32-bit, or 4-byte, numbers. As you can see, each number occupies four successive memory locations. Care must be taken when converting the values of 32-bit numbers in the four memory locations into a single decimal number. Remember that 32 bits can hold numbers between 0 and 8 600 000 000. To convert a number held in four bytes to a single decimal value the calculation is:

Total value = first byte value * 16777216 + second byte value* 65536 + third byte value * 256 + fourth byte value

```
 10 REM go
 20 REM org 23760
 30 REM First;defw 500
 40 REM defw 2000
 50 REM Second;defw 150
 60 REM defw 350
 70 REM Result;defw 0
 80 REM defw 0
 90 REM !start of program
100 REM ld hl,(First+1)
110 REM ld bc,(Second+1)
120 REM add hl,bc
130 REM ld (Result+1),hl
140 REM ld hl,(First)
150 REM ld bc,(Second)
160 REM adc hl,bc
170 REM ld (Result),hl
180 REM ret
190 REM finish
```

Figure 13.1

Although there are two different addition instructions for 16-bit numbers, there is only one subtraction instruction. It is the SBC instruction and its format is:

SBC HL,ss

where ss is one of the 16-bit registers BC, DE, HL, or SP. The effect of the SBC instruction is to subtract both the value in the 16-bit register and the value in the carry flag from the value in the HL register, leaving the result in the HL register.

The SBC instruction can be used to perform 32-bit subtractions, but before we can subtract 16-bit numbers or the first 16 bits of a 32-bit number we must ensure that the value in the carry flag is zero.

The two instructions which allow the value in the carry flag to be changed directly are, as you should remember:

SCF

which sets the value in the carry flag to one and:

CCF

which changes the value in the carry flag to the opposite of its present value. Figure 13.2 is a program to carry out 32-bit subtraction.

One final point to note is that, although the ADC and SBC instuctions affect the carry, overflow, sign, and zero flags, as you

```
 10 REM go
 20 REM org 23760
 30 REM First;defw 500
 40 REM defw 2000
 50 REM Second;defw 150
 60 REM defw 350
 70 REM Result;defw 0
 80 REM defw 0
 90 REM !start of program
100 REM ld hl,(First+1)
110 REM ld bc,(Second+1)
114 REM scf
117 REM ccf
120 REM sbc hl,bc
130 REM ld (Result+1),hl
140 REM ld hl,(First)
150 REM ld bc,(Second)
160 REM sbc hl,bc
170 REM ld (Result),hl
180 REM ret
190 REM finish
```

Figure 13.2

would expect, the 16-bit ADD instruction does not; it only affects the carry flag. This means that even for 16-bit addition the ADC instruction will normally be used, after first ensuring that the carry flag contains the value zero.

13.2 Multiple-byte numbers

We have just seen how the ADC and SBC instructions allow numbers which are multiples of 16 bits to be added and subtracted. There are, however, eight-bit versions of these instructions which allow numbers which are multiples of eight bits to be added and subtracted. The format of the instructions is:

ADC A,s
SBC A,s

where s is an eight-bit value, an eight-bit register, or the value in a memory location pointed to the HL register pair.

Figure 13.3 is a program to carry out multiple-byte addition. In this program the B register holds the number of bytes in the numbers.

```
 10 REM go
 20 REM org 23760
 30 REM !data areas
 35 REM Length;defb 3
 40 REM First;defb 12
 45 REM defb 30
 50 REM defb 100
 55 REM Second;defb 5
 60 REM defb 78
 65 REM defb 195
 70 REM Result;defb 0
 75 REM defb 0
 80 REM defb 0
 85 REM !reset carry
 90 REM scf
 95 REM ccf
100 REM !number of bytes in B
105 REM ld a,(Length)
110 REM ld b,a
115 REM !point to 1st bytes
120 REM ld de,(First+2)
125 REM ld hl,(Second+2)
130 REM ld ix,(Result+2)
135 REM !carry out addition
140 REM Loop;ld a,(de)
145 REM adc a,(hl)
150 REM ld (ix+0),a
155 REM dec de
160 REM dec hl
165 REM dec ix
170 REM djnz Loop
175 REM ret
180 REM finish
```

Figure 13.3

13.3 Binary coded decimal

Whenever we have used numbers, even if they have been input from the keyboard as decimal numbers, we have used the binary representation of the number. You have already discovered how difficult it is to convert from binary into decimal digits and then into character codes for output. Fortunately, there is another representation which can be used; it is binary coded decimal, or BCD. In the binary coded decimal representation each digit of the decimal number is represented separately. Each digit is expressed as a four-bit binary number. As each four-bit group is known as a nibble and two nibbles form a byte, any of the eight-bit registers or memory locations can hold two digits of a BCD number. Figure 13.4 shows how a four-digit decimal number could be stored in two successive memory locations.

When a nibble is used to hold a decimal digit it holds a number in the range 0 to 9, but using a straight binary representation four bits can hold numbers in the range 0 to 15, so the BCD representation does waste some space.

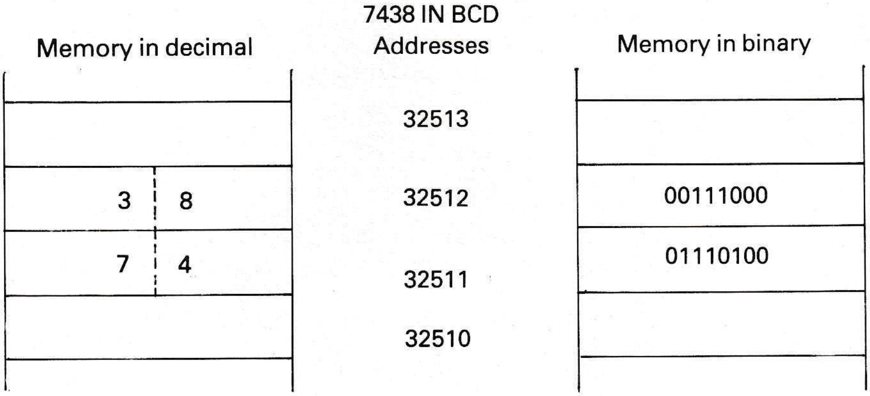


Figure 13.4

Another disadvantage with BCD numbers occurs when we carry out arithmetic. The computer is, of course, expecting pure binary numbers and gives incorrect results with BCD; this is discussed in detail in the next section. To use BCD numbers we need to be able to move the digits from memory to the accumulator and back again. Of course, we could move BCD numbers two digits at a time, or we could use shift or rotate instructions to move them one bit at a time, but both of these methods would be inconvenient. There are two instructions which give a direct move between memory and the accumulator using nibbles of data. The instructions are actually rotations involving the right-hand nibble of the accumulator and the two nibbles in a memory location. The two instructions are:

RLD	Rotate left digit
RRD	Rotate right digit

Before either instruction is executed, the HL register is used as a pointer to the required memory location. Figure 13.5 shows the operation of the instructions.

13.4 BCD arithmetic

If we take some two-digit BCD numbers and add them together

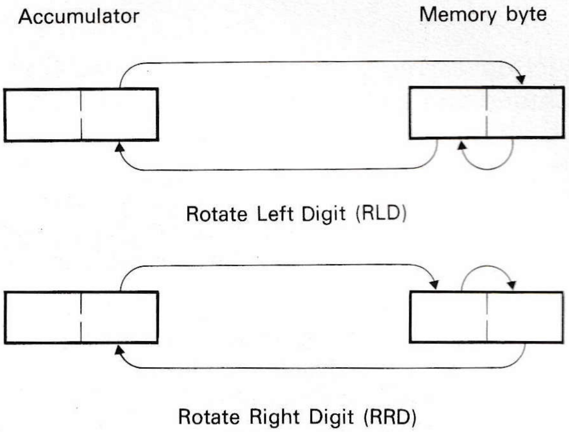


Figure 13.5

using binary arithmetic and then interpret the results as BCD numbers, we see that sometimes the result is right, and sometimes it is wrong:

```
34—00110100
51—01010001
   ————————
   10000101—85   Right answer

37—00110111
59—01011001
   ————————
   10010000—90   Wrong answer

36—00110110
55—01010101
   ————————
   10001011—8?
```

As you can see from the last answer, sometimes the result cannot even be represented as a decimal digit because the four bits represent a number greater than nine.

There are two methods used by computers to ensure that BCD arithmetic gives the right answers. One method is to provide a completely separate set of instructions for BCD arithmetic; the other method provides a way of correcting the results of binary arithmetic on BCD numbers. The Spectrum central processor uses this second method. When using BCD numbers the arithmetic instructions (ADD, ADC, SUB, or SBC) should be immediately followed by the instruction:

DAA

This instruction corrects any errors produced from the use of binary arithmetic and gives the right BCD answer. The program in Fig. 13.6 shows the input, addition, and output of two-digit BCD numbers.

The DAA instruction is also used to give correct BCD arithmetic after the instructions INC, DEC, CP, and NEG. The DAA instruction only operates on the accumulator.

```
 10 REM go
 20 REM org 23760
 30 REM !4 digit BCD addition
 35 REM Number1;defw 0
 40 REM Number2;defw 0
 45 REM Result;defw 0
 50 REM !input 1st number
 55 REM 1d h1,Number1
 60 REM call Twodig
 65 REM inc h1
 70 REM call Towdig
 75 REM !input 2nd number
 80 REM inc h1
 85 REM call Twodig
 90 REM inc h1
 95 REM call Twodig
100 REM !addition
105 REM 1d h1,Number 1
110 REM 1d a,(h1)
120 REM 1d h1,Number2
130 REM add a, (h1)
135 REM daa; !correct for BCD
140 REM 1d h1,Result
145 REM 1d (h1),a
150 REM !2nd pair of digits
155 REM 1d h1,Number 1+1
165 REM 1d a, (h1)
170 REM 1d h1,Number2+1
175 REM adc a,(h1)
180 REM daa
185 REM 1d h1,Result+1
190 REM 1d (h1),a
193 REM ret
195 REM !input subroutine
200 REM Twodig;call Cinout
205 REM sub 48
210 REM 1d (h1),a
215 REM call Cinout
```

```
220 REM sub 48
225 REM rld
230 REM ret
235 REM Cinout;call 4264
240 REM cp 208;jr z,Cinout
245 REM push af;rst 16
260 REM pop af'ret
270 REM finish
```

Figure 13.6

13.5 Other instructions

We have now looked at most of the useful assembly language instructions, but this section will look at a few more instructions which could prove useful.

Inside the Spectrum central processor, there is another set of eight-bit registers, called the auxiliary registers. These registers have the same names as the main eight-bit registers and can be used in exactly the same way as the main registers, but not at the same time.

The two sets of registers are changed over using the instruction:

EXX

This instruction swaps the two sets of registers and after its execution all instructions will refer to the second set of registers. Executing the instruction again will once more swap the registers, bringing back the first set of registers. Instead of exchanging the complete set of registers only the accumulator and flag registers can be exchanged with the instruction:

EX AF,AF′

We have already seen how the central processor can send data to the outside world using the OUT instruction. In a similar fashion the central processor can receive data from the outside world using the instruction IN. Both of these instructions have two forms. The first is:

IN A,(n)
OUT A,(n)

where A is the accumulator and n is the port number. The port number is a 16-bit number. The second form of the instructions is:

IN r,(C)
OUT r,(C)

where r is one of the eight-bit registers and the port number is in the BC register pair.

One instruction which at first appears to be of little or no use to a programmer is the NOP instruction. NOP means 'no operation' and is an instruction to do nothing. Experienced programmers find it a useful instruction. It has two main uses: although it does nothing it takes a certain amount of time to execute and it is often used to make timing loops accurate.

After you have written and assembled a long program, you will almost certainly wish to make some modifications to it. Everybody modifies their programs. Since the machine code program is stored in successive memory locations, inserting extra instructions is very difficult. One useful technique is to separate sections of program with several NOP instructions; then extra instructions or jumps to new sections of code can be easily inserted.

Finally, in this section, the last instruction we will look at is the HALT instruction. In many ways this is similar to the STOP instruction in BASIC, but on the Spectrum the program only stops until it receives a signal called an interrupt and this is sent after each television picture is produced.

This section has completed the assembly language instructions that we will be looking at in detail. There are still a small number of instructions which have not been dealt with, but they are used so rarely that they can be ignored by all but the most experienced programmers. You should now be able to write significant assembly language programs. All of the instructions which can be used with the Spectrum are shown in Appendix A.

13.6 Program

Write a program to input, add, or subtract and then output the result of BCD numbers of up to six digits.

The input will consist of a decimal number, with or without a negative sign, and either an addition sign or a minus sign followed by another decimal number. The output should look like the following:

```
    123 + 456 = 579
  772 + – 123 = 649
– 123 – 456   = – 579
```

14 ORDERING THE DATA

14.1 Sorting data

I would like to use this final chapter to look at two methods which can be used to sort data into order. There are at least 40 different methods of sorting data, but for most assembly language programs, a programmer only needs to choose between two different methods. The two methods are a simple, easily programmed method, which is relatively slow, and a more advanced method, which is more difficult to program but is much faster.

It is probably worth saying that the best method of sorting data into order in the memory of the computer is to put them into the correct order when you are entering the data into the computer. With the very fast block move and block search instructions, each data item can be put into its correct place, in the block of data, as it is input.

14.2 Bubble sort

The simplest and easiest type of sort is the *bubble sort*. The basic principle behind the bubble sort is to compare adjacent items in the list; if the items are in the wrong order they are swapped over and then the next pair of items in the list are compared. This process is repeated until all the items are in the correct order. Figure 14.1 is a flowchart for a bubble sort and Fig. 14.2 is a program to carry out a bubble sort on a list of data in a block of memory. The bubble sort is very simple to program and is relatively slow, but in assembly language it should be suitable for most applications.

14.3 Shell sort

The shell sort is in many ways only an improved form of the bubble sort, but it is significantly faster and is a good sort to use when more speed is needed.

The shell sort, like the bubble sort, makes several passes over

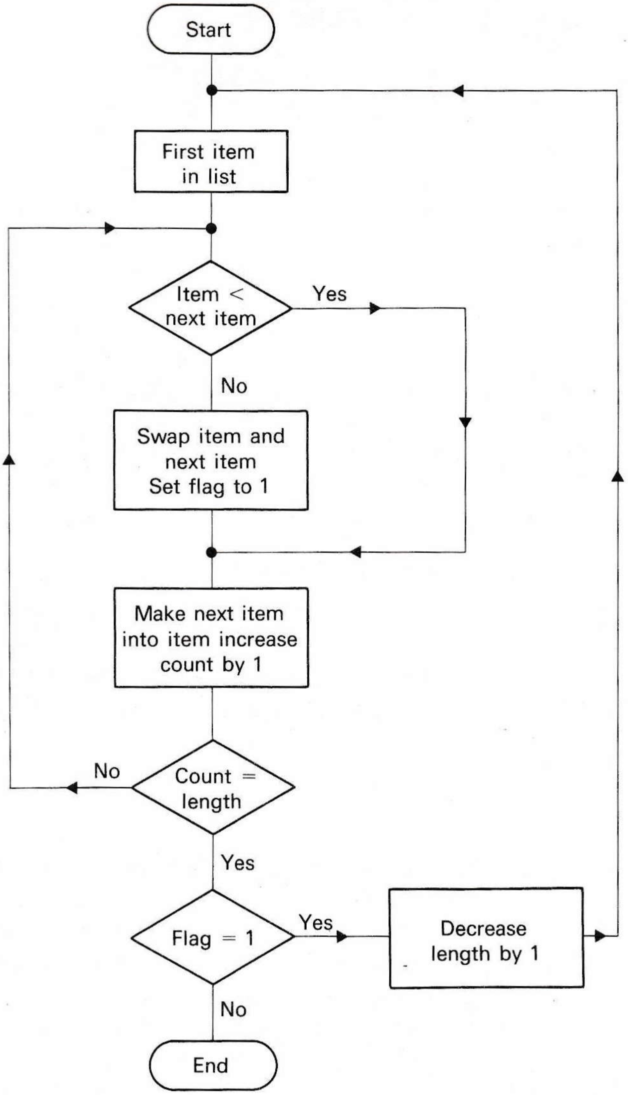

Start
First item in list
Item < next item
Yes
No
Swap item and next item Set flag to 1
Make next item into item increase count by 1
No
Count = length
Yes
Flag = 1
Yes
Decrease length by 1
No
End

Figure 14.1

the data in order to put it into order, but, unlike the bubble sort, it does not compare adjacent data items. On the first pass the first item is compared with the item in the middle of the list and then the second is compared with the one after the middle, and so on until the end of the list is reached. On the next pass the distance between the items being compared is halved, so that the first is

```
  10 REM go
  20 REM org 23760
  30 REM !Bubble sort
  35 REM Temp;defb 0
  40 REM ld e,0;!flag to show sw
aps
  41 REM !B contains length of l
ist
  43 REM !hl points to start of 
list
  45 REM Outerloop;ld a,(hl)
  50 REM inc hl;!next item
  55 REM ld c,2;!counter for ite
ms
  60 REM Innerloop;cp (hl);!test
 order
  65 REM jp m,Next
  70 REM !swap items
  75 REM ld d,a
  80 REM ld a,(hl)
  85 REM dec hl
  90 REM ld (hl),a
  95 REM inc hl
 100 REM ld (hl),d
 105 REM ld e,1;!set flag
 110 REM Next;ld a,(hl)
 115 REM inc hl
 120 REM inc c
 125 REM ld d,a
 130 REM ld a,b
 135 REM sub c;!test end of list
 140 REM jr nz,Innerloop
 145 REM ld a,e
 150 REM cp 0;!if 0 list is sort
ed
 155 REM ret z
 160 REM dec b;!reduce length by
 1
 165 REM jp Outerloop
 170 REM finish
```

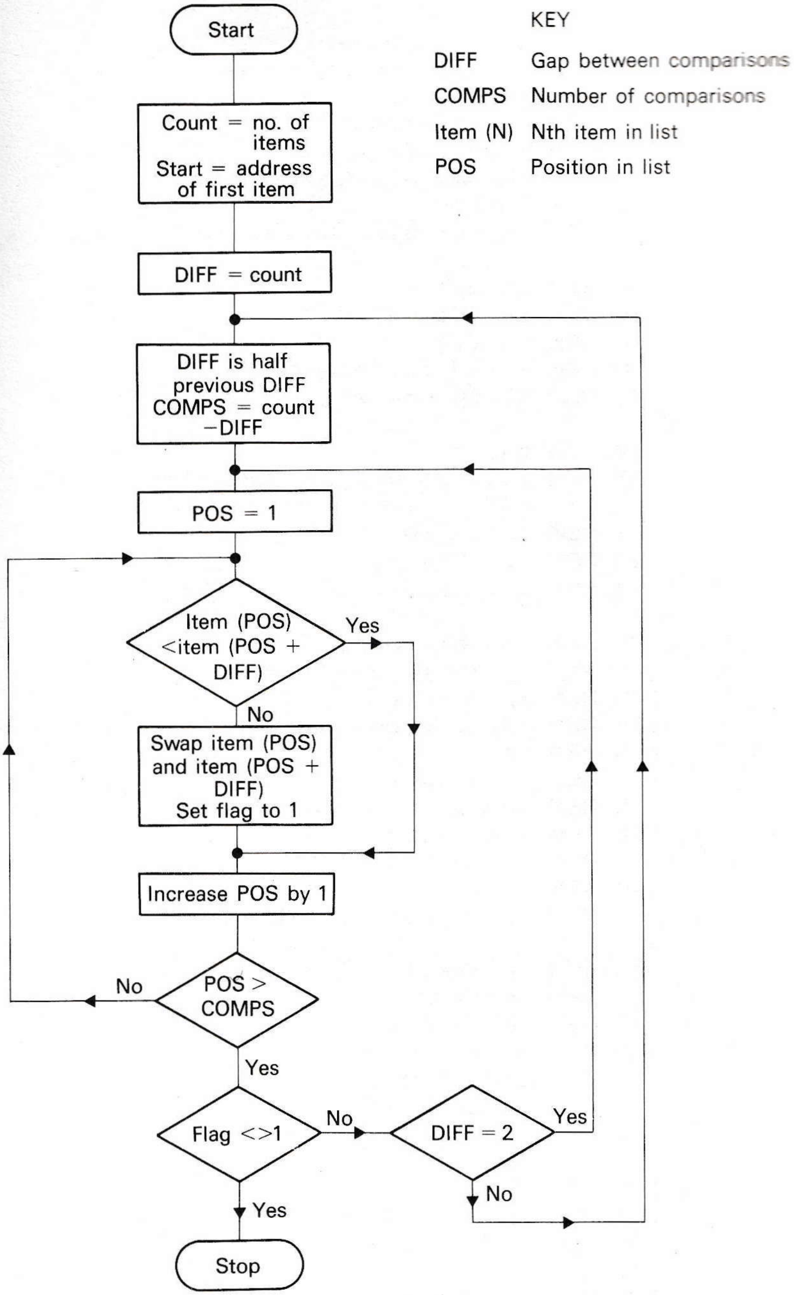
Figure 14.2

compared with the one a quarter of the way along the list. At the end of each pass the distance between the items being compared is halved until the list is sorted. Figure 14.3 shows the comparisons made on the first three passes of a list of 26 items. Figures 14.4 and 14.5 are the flowchart and program for the shell sort.

Length of list = 26 items

First pass	Second pass	Third pass	Fourth pass
Difference 13	Difference 6	Difference 3	Difference 1
1 – 14	1 – 7	1 – 4	1 – 2
2 – 15	2 – 8	2 – 5	2 – 3
3 – 16	.	.	3 – 4
.	.	.	.
.	.	.	.
.	7 – 13	5 – 8	.
12 – 25	8 – 14	6 – 9	24 – 25
13 – 26	.	.	25 – 26
	.	.	
	.	.	
	19 – 25	22 – 25	
	20 – 26	23 – 26	

Figure 14.3


Start
Count = no. of items
Start = address of first item
DIFF = count
DIFF is half previous DIFF
COMPS = count −DIFF
POS = 1
Item (POS) <item (POS + DIFF)
Yes
No
Swap item (POS) and item (POS + DIFF)
Set flag to 1
Increase POS by 1
POS > COMPS
No
Yes
Flag <>1
No
Yes
DIFF = 2
Yes
No
Stop

KEY

DIFF	Gap between comparisons
COMPS	Number of comparisons
Item (N)	Nth item in list
POS	Position in list

Figure 14.4

```
  10 REM go
  20 REM org 23760
  30 REM !Shell sort program
  31 REM !for up to 255 items
  35 REM Count;defb 0
  40 REM Flag;defb 0
  45 REM !HL points to 1st item
  50 REM ld d,0
  55 REM !B contains number of i
tems
  60 REM ld a,b
  62 REM ld (Count),a
  65 REM ld e,a
  70 REM !find difference
  75 REM Mloop;sra e;!divide by
2
  80 REM sub e
  85 REM ld b,a;!no. of comparis
ons
  87 REM xor a;!0 in A
  88 REM ld (Flag),a
  90 REM Iloop;ld a,(hl);!1st nu
mber
  95 REM add hl,de
 100 REM cp (hl);!2nd number
 105 REM jp m,Next
 110 REM !swap items
 115 REM ld c,(hl)
 120 REM ld (hl),a
 125 REM and a;!0 in carry flag
 130 REM sbc hl,de
 135 REM ld (hl),c
 140 REM ld a,1
 145 REM ld (Flag),a;!set swap f
lag
 150 REM jr Next1
 155 REM Next;and a
 160 REM sbc hl,de
 165 REM Next1;inc hl
 170 REM djnz Iloop
 175 REM ld a,(Flag)
 180 REM cp 0
 185 REM ret z
 190 REM ld a,e
 195 REM cp 2
 200 REM jp nz,Mloop
 205 REM xor a
 210 REM ld (Flag),a
 215 REM ld a,(Count)
 220 REM dec a
 225 REM ld b,a
 230 REM ld (Count),a
```

```
235 REM jp Iloop
240 REM finish
```

Figure 14.5

APPENDIX A

SUMMARY OF ASSEMBLY LANGUAGE INSTRUCTIONS

This appendix lists all of the instructions recognized by the Z80 microprocessor used in the Spectrum.

Table A.1 summarizes the effects of the instructions on the bits in the flag register. Only instructions which affect the flags are shown. The remaining tables show all of the instructions and their machine code equivalent, in decimal.

Table A.1 Flag register

	Flags					
Instruction	C	Z	P/V	S	N	H
ADD A	*	*	V	*	0	*
ADC A	*	*	V	*	0	*
SUB	*	*	V	*	1	*
SBC A	*	*	V	*	1	*
CP	*	*	V	*	1	*
NEG	*	*	V	*	1	*
AND	0	*	P	*	0	1
OR	0	*	P	*	0	1
XOR	0	*	P	*	0	0
INC m	–	*	V	*	0	*
DEC m	–	*	V	*	1	*
ADD HL	*	–	–	–	0	–
ADC HL	*	*	V	*	0	–
SBC HL	*	*	V	*	1	–
RLA, RLCA	*	–	–	–	0	0
RRA, RRCA	*	–	–	–	0	0
Rotates and shifts	*	*	P	*	0	0
RLD, RRD	–	*	P	*	0	0
DAA	*	*	P	*	–	*
CPL	–	–	–	–	1	1
SCF	1	–	–	–	0	0
CCF	*	–	–	–	0	–
IN	–	*	P	*	0	0
INI, IND, OUTI, OUTD	–	*	–	–	1	–
INIR, INDR, OTIR, OTDR	–	1	–	–	1	–
LDI, LDD	–	–	*	–	0	0

Instruction	Flags C	 Z	 P/V	 S	 N	 H
LDIR, LDDR	–	–	0	–	0	0
CPI, CPIR, CPD, CPDR	–	*	*	*	1	–
BIT	–	*	–	–	0	1
NEG	*	*	V	*	1	*

The following notation is used in the table:

m	Eight-bit operands only
*	Flag is affected
–	Flag is unaffected
0	Flag is set to 0
1	Flag is set to 1
V	Flag shows overflow
P	Flag shows parity

Table A.2 Eight-bit load instructions—LD d,s

	Source of data										
Destination	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
A	127	120	121	122	123	124	125	126	221 126 d	253 126 d	63 n
B	71	64	65	66	67	68	69	70	221 70 d	253 70 d	6 n
C	79	72	73	74	75	76	77	78	221 78 d	253 78 d	14 n
D	87	80	81	82	83	84	85	86	221 86 d	253 86 d	22 n
E	95	88	89	90	91	92	93	94	221 94 d	253 94 d	30 n
H	103	96	97	98	99	100	101	102	221 102 d	253 102 d	38 n
L	111	104	105	106	107	108	109	110	221 110 d	253 110 d	46 n
(HL)	119	112	113	114	115	116	117				54 n
(IX+d)	221 119 d	221 112 d	221 113 d	221 114 d	221 115 d	221 116 d	221 117 d				
(IY+d)	253 119 d	253 112 d	253 113 d	253 114 d	253 115 d	253 116 d	253 117 d				

	Source of data					
Destination	(BC)	(DE)	(nn)	R	I	A
A	10	26	58 n n	237 95	237 87	
(BC)						2
(DE)						18
(nn)						50 n n
R						237 79
I						237 71

Table A.3 Sixteen-bit load instructions

	Source of data	
	nn	(nn)
BC	1, n, n	237,75, n, n
DE	17, n, n	237, 91, n, n
HL	33, n, n	42, n, n
SP	49, n, n	237, 123, n, n
IX	221, 33, n, n,	221, 42, n, n
IY	253, 33, n, n	253, 42, n, n

	Source of data					
	BC	DE	HL	SP	IX	IY
(nn)	237 67 n n	237 83 n n	34 n n	237 115 n n	221 34 n n	253 34 n n

Table A.4 PUSH and POP instructions

	AF	BC	DE	HL	IX	IY
					221	253
PUSH	245	197	213	229	229	229
					221	253
POP	241	193	209	225	225	225

Table A.5
Exchange instructions

EXX	217
EX AF,AF′	8
EX DE,HL	235
EX (SP),HL	227
EX (SP),IX	221, 227
EX (SP),IY	253, 227

Table A.6
Block instructions

LDI	237, 160
LDIR	237, 176
LDD	237, 168
LDDR	237, 184
CPI	237, 161
CPIR	237, 177
CPD	237, 169
CPDR	237, 185

Table A.7
General arithmetic

DAA	39
CPL	47
NEG	237, 68
CCF	63
SCF	55

Table A.8 Eight-bit arithmetic and logic

	Source of data										
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
ADD	135	128	129	130	131	132	133	134	221, 134, d	253, 134, d	198, n
ADC	143	136	137	138	139	140	141	142	221, 142, d	253, 142, d	206, n
SUB	151	144	145	146	147	148	149	150	221, 150, d	253, 150, d	214, n
SBC	159	152	153	154	155	156	157	158	221, 158, d	253, 158, d	222, n
AND	167	160	161	162	163	164	165	166	221, 166, d	253, 166, d	230, n
XOR	175	168	169	170	171	172	173	174	221, 174, d	253, 174, d	238, n
OR	183	176	177	178	179	180	181	182	221, 182, d	253, 182, d	246, n
CP	191	184	185	186	187	188	189	190	221, 190, d	253, 190, d	254, n
INC	60	4	12	20	28	36	44	52	221, 52, d	253, 52, d	
DEC	61	5	13	21	29	37	45	53	221, 53, d	253, 53, d	

Table A.9 Sixteen-bit arithmetic

	Source of data					
	BC	DE	HL	SP	IX	IY
ADD HL	9	25	41	57		
ADD IX	221 9	221 25		221 57	221 41	
ADD IY	253 9	253 25		253 57		253 41
ADC	237 74	237 90	237 106	237 122		
SBC	237 66	237 82	237 98	237 114		
INC	3	19	35	51	221 35	253 35
DEC	11	27	43	59	221 43	253 43

Table A.10 Rotate and shift instructions

	Source of data									
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
RLC(A)									221	253
		203	203	203	203	203	203	203	203	203
	7	0	1	2	3	4	5	6	d	d
									6	6
RRC(A)									221	253
		203	203	203	203	203	203	203	203	203
	15	8	9	10	11	12	13	14	d	d
									14	14
RL(A)									221	253
		203	203	203	203	203	203	203	203	203
	23	16	17	18	19	20	21	22	d	d
									22	22
RR(A)									221	253
		203	203	203	203	203	203	203	203	203
	31	24	25	26	27	28	29	30	d	d
									30	30
SLA									221	253
	203	203	203	203	203	203	203	203	203	203
	39	32	33	34	35	36	37	38	d	d
									38	38
SRA									221	253
	203	203	203	203	203	203	203	203	203	203
	47	40	41	42	43	44	45	46	d	d
									46	46
SRL									221	253
	203	203	203	203	203	203	203	203	203	203
	63	56	57	58	59	60	61	62	d	d
									62	62
RLD								237		
								111		
RRD								237		
								103		

Table A.11 Bit instructions

The BIT instruction

Bit number	0	1	2	3	4	5	6	7
A	203 71	203 79	203 87	203 95	203 103	203 111	203 119	203 127
B	203 64	203 72	203 80	203 88	203 96	203 104	203 112	203 120
C	203 65	203 73	203 81	203 89	203 97	203 105	203 113	203 121
D	203 66	203 74	203 82	203 90	203 98	203 106	203 114	203 122
E	203 67	203 75	203 83	203 91	203 99	203 107	203 115	203 123
H	203 68	203 76	203 84	203 92	203 100	203 108	203 116	203 124
L	203 69	203 77	203 85	203 93	203 101	203 109	203 117	203 125
(HL)	203 70	203 78	203 86	203 94	203 102	203 110	203 118	203 126
(IX+d)	221 203 d 70	221 203 d 78	221 203 d 86	221 203 d 94	221 203 d 102	221 203 d 110	221 203 d 118	221 203 d 126
(IY+d)	253 203 d 70	253 203 d 78	253 203 d 86	253 203 d 94	253 203 d 102	253 203 d 110	253 203 d 118	253 203 d 126

The RES instruction

Bit number	0	1	2	3	4	5	6	7
A	203 135	203 143	203 151	203 159	203 167	203 175	203 183	203 191
B	203 128	203 136	203 144	203 152	203 160	203 168	203 176	203 184
C	203 129	203 137	203 145	203 153	203 161	203 169	203 177	203 185
	203	203	203	203	203	203	203	203

The RES instruction Bit number	0	1	2	3	4	5	6	7
D	130	138	146	154	162	170	178	186
	203	203	203	203	203	203	203	203
E	131	139	147	155	163	171	179	187
	203	203	203	203	203	203	203	203
H	132	140	148	156	164	172	180	188
	203	203	203	203	203	203	203	203
L	133	141	149	157	165	173	181	189
	203	203	203	203	203	203	203	203
(HL)	134	142	150	158	166	174	182	190
	221	221	221	221	221	221	221	221
(IX+d)	203	203	203	203	203	203	203	203
	d	d	d	d	d	d	d	d
	134	142	150	158	166	174	182	190
	253	253	253	253	253	253	253	253
(IY+d)	203	203	203	203	203	203	203	203
	d	d	d	d	d	d	d	d
	134	142	150	158	166	174	182	190

The SET instruction Bit number	0	1	2	3	4	5	6	7
	203	203	203	203	203	203	203	203
A	199	207	215	223	231	239	247	255
	203	203	203	203	203	203	203	203
B	192	200	208	216	224	232	240	248
	203	203	203	203	203	203	203	203
C	193	201	209	217	225	233	241	249
	203	203	203	203	203	203	203	203
D	194	202	210	218	226	234	242	250
	203	203	203	203	203	203	203	203
E	195	203	211	219	227	235	243	251
	203	203	203	203	203	203	203	203
H	196	204	212	220	228	236	244	252

The SET instruction

Bit number	0	1	2	3	4	5	6	7
L	203 197	203 205	203 213	203 221	203 229	203 237	203 245	203 253
(HL)	203 198	203 206	203 214	203 222	203 230	203 238	203 246	203 254
(IX+d)	221 203 d 198	221 203 d 206	221 203 d 214	221 203 d 222	221 203 d 230	221 203 d 238	221 203 d 246	221 203 d 254
(IY+d)	253 203 d 198	253 203 d 206	253 203 d 214	253 203 d 222	253 203 d 230	253 203 d 238	253 203 d 246	253 203 d 254

Table A.12 Jump, call, and return instructions

	Condition								
Instruction	None	C	NC	Z	NZ	PE	PO	M	P
JP nn	195 n n	218 n n	210 n n	202 n n	194 n n	234 n n	226 n n	250 n n	242 n n
JR n	24 n	56 n	48 n	40 n	32 n				
JP (HL)	233								
JP (IX)	221 233								
JP (IY)	253 233								
CALL nn	205 n n	220 n n	212 n n	204 n n	196 n n	236 n n	228 n n	252 n n	244 n n
RET	201	216	208	200	192	232	224	248	240
DJNZ	16 n								
RETI	237 77								
RETN	237 69								

Table A.13
Restart instructions

RST 0	199
RST 8	207
RST 16	215
RST 24	223
RST 32	231
RST 40	239
RST 48	247
RST 56	255

Table A.14 Input and output instructions

		Register						
Instruction	A,(n)	A,(C)	B,(C)	C,(C)	D,(C)	E,(C)	H,(C)	L,(C)
IN	219	237	237	237	237	237	237	237
	n	120	64	72	80	88	96	104
OUT	211	237	237	237	237	237	237	237
	n	121	65	73	81	89	97	105
INI	237, 162							
INIR	237, 178							
IND	237, 170							
INDR	237, 186							
OUTI	237, 163							
OTIR	237, 179							
OUTD	237, 171							
OTDR	237, 187							

Table A.15
Miscellaneous instructions

NOP	0
HALT	118
DI	243
EI	251
IMO	237, 70
IM1	237, 86
IM2	237, 94

APPENDIX B
THE ZX SPECTRUM MACHINE CODE ASSEMBLER

B.1 Using an assembler

All of the programs in this book have been produced using the ZX Spectrum Machine Code Assembler program from ACS software. This appendix describes the use of this particular assembler program but the ideas will generally be applicable to any Spectrum assembler program.

The assembler program is, itself, a machine code program which is loaded into the top of memory by an accompanying BASIC program. Once the machine code program is loaded the BASIC program is automatically deleted.

When needed, the assembler program is called into use by a command from BASIC, RANDOMIZE USR 26 000 (for the 16K Spectrum), or RANDOMIZE USR 58 000 (for the 48K Spectrum).

We have already seen that two areas of memory are used when translating assembly language programs. One area is used to store the assembly language program and the other is used for the machine code program. During program development it is easiest to set aside an area of memory in a REM statement, at the beginning of a BASIC program, for the machine code program. Nearly all of the programs shown in this book have been developed using this method. The accompanying BASIC program must start with a REM statement containing as many characters as there will be memory bytes used by the machine code program. You will not normally know how many bytes are required before you have actually translated your assembly language program. An estimate can be made by allowing two bytes for each assembly language instruction.

B.2 ZX Spectrum assembler

The ZX Spectrum Machine Code Assembler also uses the BASIC program area to store the assembly language instructions. All instructions are written in a BASIC program in REM statements. A glance at some of the programs in the body of the book will illustrate how programs are prepared. More than one instruction can be written in a single REM statement, provided that the instructions are separated by a semi-colon (;).

The assembler recognizes all of the Z80 instructions shown in Appendix A and also several directives, which are listed later. Directives are instructions which do not translate into machine code instructions but are used to give instructions to the assembler program. Directives are often called pseudo-ops because they look like the instructions which translate into machine code.

Throughout this book all instructions included in the text have been printed in upper case (capital) letters so that they can easily be distinguished from the remainder of the text. When using the ZX Spectrum Assembler all instructions must be entered in *lower case* letters, as shown in the back of the Sinclair Spectrum manual.

Numbers used with the ZX Spectrum Assembler can be decimal or hexadecimal. Hexadecimal numbers are preceded by a dollar sign (e.g., $1AB5). Labels can be used to refer to memory locations and the assembler will automatically convert the label into the correct memory address. Labels with the ZX Spectrum Assembler can be of any length but the first character must be a capital letter. The only other restrictions on labels are that they must not contain the characters ')' or '+'. When used at the beginning of an instruction as a pointer to that instruction (or directive), the label is treated as a separate instruction and is followed by a semi-colon.

There are three types of error message which can be given during assembly. If there are any errors in the GO, FINISH, or ORG directives an error message will be given before assembly starts. An incorrect instruction will be shown by a flashing message which gives the line number, the number of the instruction in the line, and the type of instruction. This should enable you to find the instruction containing the error easily; unfortunately it is not always as easy to find the error itself. Finally, you can get one of the Sinclair error messages. The possible messages are:

B	Integer out of range—a number outside the allowed range
2	Variable not found—reference to a non-existent label
Q	Parameter error—mistyped instructions
6	Number too big—relative jump displacement out of range

B.3 Directives

go—all ZX Spectrum Assembler programs must start with this instruction. It must be in a REM statement on its own.

finish—this must be the last instruction in any program. Again it must be in a REM statement on its own.

org—this instruction tells the assembler which location in memory to use for the start of the machine code program. On a standard 16K or 48K Spectrum, org 23 760 will load the machine code program into the first free byte of a REM statement at the beginning of a program. **org** should always be the second instruction of the assembly language program, but it may also be used in the middle of a program to make the assembler jump to a new location in memory.

defb—allows one or more bytes of memory, starting at the current assembly address, to be set to defined values in the range 0 to 255.

defw—allows one word (two bytes) of memory to be set to a defined value in the range 0 to 65 535.

defs—allows a string of characters to be put into memory.

equ—used for assigning a label to a memory location. Useful when the address of the location is outside the machine code program. It can be used to link sections of machine code together.

APPENDIX C
HEXADECIMAL–DECIMAL CONVERSION TABLES

This book has been written using mainly decimal numbers, because they are usually easier for people to understand. There are, however, occasions when the use of binary, or the related hexadecimal, numbers have more meaning—in particular, when considering bit patterns in a register or memory location, or when converting a 16-bit memory address into two eight-bit numbers. The following tables provide conversion between hexadecimal and decimal. Table C.1 gives the conversion for hexadecimal numbers up to FF or 255 in decimal and Table C.2 used in conjunction with the first table covers numbers up to FFFF or 65 535 in decimal.

Table C.1
Conversion of hexadecimal numbers up to FF, or 255 in decimal

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
20	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
30	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
40	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
50	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
60	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
70	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
80	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
90	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
AO	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
BO	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
CO	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
DO	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
EO	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
FO	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Table C.2
Conversion of hexadecimal numbers up to FFFF, or 65 535 in decimal (in conjunction with Table C.1)

Hexadecimal	Decimal	Hexadecimal	Decimal
100	256	1000	4096
200	512	2000	8192
300	768	3000	12288
400	1024	4000	16384
500	1280	5000	20480
600	1536	6000	24576
700	1792	7000	28672
800	2048	8000	32768
900	2304	9000	36864
A00	2560	A000	40960
B00	2816	B000	45056
C00	3072	C000	49152
D00	3328	D000	53248
E00	3584	E000	57344
F00	3840	F000	61440

APPENDIX D
HAND ASSEMBLY

D.1 General method

A program written in assembly language has to be translated into machine code before it can be run on a computer. The easiest way to carry out this translation is to use an assembler program. For anybody who is interested in writing any but the smallest of programs, the use of an assembler is strongly recommended. The other method of translating to machine code is to use tables of instructions and convert the program by hand.

To translate from assembly language to machine code, each instruction must be looked up in tables of instructions, such as the tables in Appendix A, in order to find the numeric form of the instruction. This number can be in binary, hexadecimal, or decimal; if the number is in binary it is usual to convert it to hexadecimal, which is easy, or to decimal.

After the program has been translated into a list of numbers it must be loaded into the computer's memory. Decimal numbers can be POKEd into memory using a very simple program. Hexadecimal numbers have to be converted to decimal before they can be POKEd into memory. Of course, before you enter your program into memory you must decide where in memory it will go, and perhaps enter instructions to set aside an area of memory for the program.

D.2 Addresses and data

As well as converting all your instructions into a numeric form, you must also convert all of your data into numbers in the same number base as your instructions. The computer only distinguishes between data and instructions by knowing which should be in the next memory location. For example, if the starting address of a program is location 32 000, the computer will take the number in memory location 32 000 as an instruction. If the number in location 32 000 translates into an instruction which should be followed by data, such as instructions LD A,n then the computer will take the number in location 32 001 as the value of n.

One of the biggest problems with hand assembly arises with the translation of 16-bit numbers. These must be entered into the computer as two eight-bit numbers with the right-hand eight bits entered before the left-hand eight bits. Figure D.1 shows the conversion of the decimal number 32 000 into two eight-bit numbers and converted to decimal. Using this example the instruction CALL 32 000 would be put into three successive memory locations as 205, 0, 125.

$$\begin{aligned}32000 &= 0111110100000000\text{B}\\ &= 01111101/00000000\\ &= 125 \qquad\qquad 0\\ 32000 &= 0, 125\end{aligned}$$

Figure D1

D.3 Jump instructions

There are two types of jump instructions to be considered: absolute jumps (JP instructions) and relative jumps (JR instructions). The method of translation is the same whether the jumps are conditional or unconditional.

Absolute jumps are followed by the actual address of the memory location which holds the instruction which follows a jump. If you are jumping further on in the program you will have to wait until you have translated the program up to that instruction before you can find the required address. Remember that the bytes of the address will be reversed, as shown in the last section.

Relative jumps cause the most problems when assembling programs by hand. If your program does not work properly it will probably be worth recalculating your relative jumps. The data for a relative jump instruction are the numbers of memory locations from the jump instruction to the next instruction minus two. If it is a jump back in the program this will give a negative number of locations, which is expressed as an eight-bit 'two's complement' number. The two is subtracted from the displacement because when an instruction is run the program counter is already pointing to the next instruction. The program in Fig. D.2 illustrates the translations of two relative jumps.

D.4 Bit instructions

Because the instructions for testing, setting, and resetting individual bits in a register depend upon both the register and the

```
Assembly language        Machine code   Comment
Loop;  1d a,(hl)             126
       cp0                   254
                               0
       jr z,Endit             40        Jump forward 5
                                        locations
                               3        (5 – 2)
       inc hl                 35
       jr loop                24        Jump back 6
                                        locations
                             248        (–6 –2 in two's
                                        complement)
Endit; ret                   201
```

Figure D.2

bit number, great care should be taken to ensure that the correct instruction code is used. All of the bit instructions have the code 203 as the first byte of the instruction.

D.5 Index registers

The index register instructions look more complex than most instructions because many of them are three- or four-byte instructions. They can, in fact, be greatly simplified if it is noted that they have the same machine code instructions as the equivalent instructions for the HL register pair, except that instructions using the IX register are prefixed with the byte 221 and the IY register instructions are prefixed with the byte 253.

Both index registers, when they are used as memory pointers, must include a byte which shows the displacement from the location contained in the index register, even if the displacement is zero.

APPENDIX E
CHARACTER CODES

The table in this appendix, Table E.1, lists the characters and their codes which can be input from the keyboard. Apart from the graphics characters these are also the characters which are most used for output.

Table E.1

32	(space)	80	P
33	!	81	Q
34	''	82	R
35	≠	83	S
36	$	84	T
37	%	85	U
38	&	86	V
39	'	87	W
40	(	88	X
41	)	89	Y
42	*	90	Z
43	+	91	[
44	,	92	/
45	−	93	]
46	.	94	↑
47	/	95	—
48	0	96	£
49	1	97	a
50	2	98	b
51	3	99	c
52	4	100	d
53	5	101	e
54	6	102	f
55	7	103	g
56	8	104	h
57	9	105	i
58	:	106	j

Table E.1 cont.

59	;	107	k
60	<	108	l
61	=	109	m
62	>	110	n
63	?	111	o
64	@	112	p
65	A	113	q
66	B	114	r
67	C	115	s
68	D	116	t
69	E	117	u
70	F	118	v
71	G	119	w
72	H	120	x
73	I	121	y
74	J	122	z
75	K	123	{
76	L	124	|
77	M	125	}
78	N	126	~
79	O	127	©

APPENDIX F
PRINT CONTROL CHARACTERS

Code	Effect
6	Print Comma (move half screen)
8	Backspace
13	Next Line
16	INK
17	PAPER
18	FLASH
19	BRIGHT
20	INVERSE
21	OVER
22	AT
23	TAB

APPENDIX G
ROM SUBROUTINES

G.1 The ROM program

When you switch on your Spectrum, a program starts running immediately; this is the program in ROM (Read Only Memory) and occupies the first 16K of the available memory. The purpose of this program is to enable the microprocessor to communicate with the various input and output devices used by the Spectrum and to enable BASIC programs to be entered and run. In the standard Spectrum system, the main inputs are from the keyboard or the cassette player and outputs are sent to the television display, the loudspeaker, the cassette recorder, and the printer.

The ROM program is a machine code program and is written as a series of subroutines. This means that these routines are available for use in your assembly language programs. This appendix will list some of the easier routines to use and show you how they can be used.

G.2 Print a character

The character whose code is in the A register can be displayed on the current channel by the single instruction:

```
RST 16
```

To print to the top part of the screen, the correct channel must be opened first and then the following program segment will print a character to the top part of the screen:

```
LD A,2
CALL 5633
LD A,Code ; ! character to be printed
RST 16
```

As shown earlier, this routine can also be used to change the current print position and the temporary colours by using the print control codes given in Appendix F.

G.3 Clear screen

The whole screen can be cleared with the following program segment:

```
LD A,2
CALL 5633
CALL 3435
```

There is another routine which can be used to clear just part of the screen. This routine clears a given number of lines, counting from the bottom of the screen:

```
LD B,Lines ; ! Number of lines to be cleared
CALL 3652
```

G.4 Screen scrolling

The screen can be made to scroll automatically by repeatedly loading the system variable SCR CT with a number larger than one—255 is probably the best choice. Use the instructions:

```
PUSH HL
LD HL,23692
LD (HL),255
POP HL
```

There is a routine in the ROM which enables you to set the number of lines which will be scrolled. Again the number of lines is counted from the bottom of the screen, the value loaded into the B register being one less than the number of lines to be scrolled:

```
LD B,Lines
CALL 3584
```

G.5 Border colour

The border colour is changed by putting the number of the required colour in the A register and then calling a ROM subroutine:

```
LD A,Colour
CALL 8859
```

G.6 Screen colours

The colour variables are stored as bytes in the attribute file and the system variables ATTR-P, ATTR-T, MASK-P, and MASK-T. Usually the Spectrum ROM routines use the temporary colour values but some, like the clear screen routine, use the permanent values.

The permanent attributes can be set by changing the appropriate bits of system variable ATTR-P; this is at location 23 693. The attributes are stored as follows:

Bits 0–2	INK colour
Bits 3–5	PAPER colour
Bit 6	Set for BRIGHT
Bit 7	Set for FLASH

Once the colours have been set, as either permanent or temporary items, the following routines can be used. To copy permanent values to the temporary systems variables use the instruction:

```
CALL 3405
```

and to copy temporary values to permanent use:

```
CALL 7341
```

G.7 Keyboard input

The simplest routine to use is the one which checks the keyboard until a key press is pressed. If a key is pressed a value is put in the DE register pair. This routine can be used by the instruction:

```
CALL 654
```

The main character input routine has been given in Chapter 6.

G.8 Sound

The routine to send a single note to the loudspeaker can be called with the instruction:

```
CALL 949
```

Before the instruction is used the HL register is loaded with a value which sets the pitch of the note and the DE register is loaded with a value which determines the length of the note.

G.9 The printer

There are two routines in the ROM which can be used with the printer. The simplest is the COPY routine which copies the screen to the printer. This is used by the instruction:

CALL 3691

The contents of the printer's buffer are passed to the printer by the instruction:

CALL 3789

G.10 Graphics

The graphics routines for PLOT and DRAW can be called from the ROM to give high-resolution graphics easily. The PLOT routine requires the x position to be loaded into the C register and the y position into the B register before calling the subroutine with:

CALL 8927

The DRAW is more complicated because we can use both positive and negative values of x and y. Before the subroutine is called the B and C registers should hold the absolute values of y and x respectively and D and E registers should hold the signs of x and y respectively. If x is positive, D will contain the value 1; if it is negative, D will contain the value −1; and finally, if x is zero, D will hold a zero value. To call the routine the instruction is:

CALL 9402

INDEX

Accumulator, 6
ADC, 116–118
ADD, 28, 116–118
Address, 12
Addressing modes, 17, 61–64
AND, 95
ASCII codes, 47
Assembler, 18, 144–146
Assembling, 17–19
Assembly language, 13–15
Attribute file, 99

Base, 7
BCD (*see* Binary coded decimal)
Binary coded decimal, 119–122
Binary numbers, 7–10
Bit, 5
BIT, 93
Block moves, 81–84
Block search, 101–102
Bubble sort, 125
Byte, 10

CALL, 21
Carry, 51–55
CCF, 53, 117
Central processing unit, 4
Clear screen, 86
Comparisons, 37–39
Conditional jumps, 35–37
CP, 38
CPD, 102
CPDR, 101
CPI, 102
CPIR, 101
CPL, 95

DAA, 122
DEC, 26
DEFB, 40
DEFS, 75
DJNZ, 31, 68–69
Directive, 31
Display file, 78

EQU, 55
EX, 57, 123
EXX, 123

FINISH, 39
Flag register, 6, 34–35, 132–133

GO, 39

HALT, 124
Hand assembly, 149–151
Hexadecimal numbers, 7–10
High level language, 2

IN, 59, 123
INC, 26
Index registers, 6, 102–103

JP, 31, 33–34
JR, 34
Jump tables, 110

Keyboard input, 46–47, 58–59

Labels, 21, 31
LD, 24–28, 57, 74
LDD, 83
LDDR, 83
LDI, 83
LDIR, 82
Logical operators, 95
Look-up tables, 103–110
Loops, 66–70
Low level language, 2

Machine code, 13–14, 17
Memory, 11–12
Memory mapping, 40
Memory pointer, 26–28
Messages, 74–76
Microprocessor, 4–6
Multiplication, 89–90
Music, 61

NEG, 50
Nibble, 119
NOP, 124
NOT, 95
Number input, 47–50

Operand, 17, 62
OR, 95
ORG, 39
OUT, 86, 123
Overflow, 51–55

Packed data, 96–97
Packing, 97–98
Pointer, 26–28
POP, 73
Port, 59
Program, 2
Pseudo-operation, 39–40
PUSH, 73

RAM, 12
RAMTOP, 15, 20
Random numbers, 113–114
Register, 5–6
RES, 94
RET, 19, 21–23
RL, 92
RLA, 91
RLC, 92
RLCA, 91
RLD, 120
ROM, 12
Rotations, 90–92
RR, 92
RRA, 91
RRC, 92
RRCA, 91
RRD, 120
RST 16, 41–44

SBC, 117–118
SCF, 53, 117
Screen border, 86
Screen output, 40–44
Scrolling, 84–86, 94
SET, 94
Shell sort, 125–127
Shifts, 88–90
Sign flag, 34–37
Signed numbers, 10–11
Single step, 110
Stack, 70–74
Status register, 6
SUB, 28
Subroutine, 20–23, 155–158

Two's complement, 10–11, 51

Unconditional jump, 33–34
Unpacking, 97–98
Unsigned number, 10
USR function, 19

XOR, 95

ZX Spectrum assembler, 18, 144–146
Zero flag, 34–37

Cassette Software . . .

ZX SPECTRUM MACHINE CODE ASSEMBLER

All the programs in this book were assembled using the McGraw-Hill **ZX Spectrum Machine Code Assembler.**
This is a full-function assembler, for the 16K or 48K machine, assembling all standard Zilog Z80 mnemonics, plus a useful range of pseudo instructions. If you want a first-class assembler, that will run the programs in this book with the greatest possible ease and convenience, you *need* this software!

Star Features:

★ Uses the Spectrum editing system, so there are no new keys to learn.
★ Locates assembled code anywhere in memory.
★ Saves assembled or unassembled code on tape.
★ Outputs assembled code, memory locations, and hex code to screen (with colour) or printer — either the ZX printer, or a 'real' printer via a suitable interface.
★ Full size 12-page manual.
★ Free McGraw-Hill Software Support Service available if you have problems.
★ Silver pack to match this book.

JUST £7.95 INC. V.A.T.
(07 084717 7)

The next step . . .

ASSEMBLY LANGUAGE FOR ARCADE GAMES and other Fast Spectrum Programs

by Stuart Nicholls

Reveals professional programming techniques for Assembly Language programmers . . . and explains how those fast arcade games are really written!

£7.95
(January 1984)
(07 084729 0)

McGraw-Hill books and software should be available where you bought this book, but in case of difficulty, send direct to:

McGraw-Hill Book Company (UK) Limited
Shoppenhangers Road, Maidenhead, Berkshire

Machine code for action

Machine-code programs run very quickly, literally hundreds of times faster than the equivalent BASIC. And they are very powerful, enabling your computer to do things undreamed of by its manufacturer!

The ZX Spectrum uses a Z80 processor, and machine-code programs are written in Z80 assembly language. This book is a complete course in Z80 assembly language, right from the beginning. Written by an experienced teacher and programmer, it is ideal for anyone with a knowledge of BASIC and contains many complete - and useful - programs.

Assembly software

The assembly-language listings in this book can of course be hand-assembled into machine-code, but for speed and accuracy you should use the McGraw-Hill *ZX Spectrum Machine Code Assembler* program, available on cassette for the 16K or 48K Spectrum. All the program listings have been specially written to match this software. See inside for details.

McGRAW-HILL Book Company (UK) Limited
MAIDENHEAD · BERKSHIRE · ENGLAND
07 084705 3